Mixin' Up the ML Module System

Derek Dreyer Andreas Rossberg

Max Planck Institute for Software Systems (MPI-SWS)

{dreyer,rossberg}@mpi-sws.mpg.de

Abstract

ML modules provide hierarchical namespace management, as well as fine-grained control over the propagation of type information, but they do not allow modules to be broken up into mutually recursive, separately compilable components. Mixin modules facilitate recursive linking of separately compiled components, but they are not hierarchically composable and typically do not support type abstraction. We synthesize the complementary advantages of these two mechanisms in a novel module system design we call MixML.

A MixML module is like an ML structure in which some of the components are specified but not defined. In other words, it unifies the ML structure and signature languages into one. MixML seamlessly integrates hierarchical composition, translucent ML-style data abstraction, and mixin-style recursive linking. Moreover, the design of MixML is clean and minimalist; it emphasizes how all the salient, semantically interesting features of the ML module system (as well as several proposed extensions to it) can be understood simply as stylized uses of a small set of orthogonal underlying constructs, with mixin composition playing a central role.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Recursion, Abstract data types, Modules; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Design, Theory

Keywords Type systems, ML modules, mixin modules, abstract data types, recursive modules, hierarchical composability

1. Introduction

ML modules and *mixin modules* are two well-known and influential mechanisms for modular programming that have largely complementary advantages and disadvantages. In this paper, we show how to synthesize some of the defining aspects of these mechanisms in the design of a novel module system we call MixML.

We begin by reviewing some of the main features and drawbacks of ML modules and mixin modules.

1.1 ML Modules

Proposed originally by MacQueen [23] in the mid-1980s and developed further by Harper, Leroy, and many others [15, 20, 10], the ML module system offers powerful support for flexible program construction, data abstraction, and code reuse. In ML, *structures* provide namespace management, *signatures* describe module interfaces, *functors* enable the definition of generic modules, and *opaque signature ascription* (aka *sealing*) lets one hide the implementation details of a module behind an interface.

One important feature of ML modules is that they are *hierarchically composable*. Structures may contain other structures as components, and thus be used to build hierarchical namespaces. Another important feature is that ML modules may contain both *dynamic* components, defined by core-ML terms, and *static* components, defined by core-ML types. The packaging together of types and terms, along with the opaque sealing construct, allows modules to express *abstract data types*. Furthermore, signatures are *translucent* [15], *i.e.,* they can specify type components of modules either abstractly or transparently. Translucency gives the programmer fine-grained control over the propagation of type information.

However, one major limitation of ML modules (at least traditionally) is that they cannot be defined recursively, thus inhibiting the decomposition of mutually recursive functions and data types into modular components. Consequently, in the last decade, there have been several proposals for extending ML with recursive modules [6, 33, 22, 26, 7]. While the existing proposals address a variety of interesting issues, such as the interaction of recursion and data abstraction [6, 7], none of them provides adequate support for something we view as a central design goal: *separate compilation*. Half the motivation for recursive modules is the desire to break big modules into smaller components that can be compiled independently of one another and linked with multiple different implementations of the other components. Yet, except in restricted cases, this functionality is not accounted for by any of the existing proposals.

We believe that the reason existing proposals have failed to support general separate compilation of mutually recursive modules is that ML's traditional means of supporting separate compilation and hierarchical (*i.e.,* non-recursive) linking—namely, *functors*—do not scale well to the recursive case. The body of a functor (which defines its *exports*) may depend on its argument (which specifies its *imports*), but not vice versa. In the context of recursive modules, however, the import specifications of a separately-compiled module may in general need to refer recursively to abstract type components provided in its exports. Unfortunately, it is not obvious how to generalize the functor mechanism in a simple way in order to permit the argument to depend on the result.

1.2 Mixin Modules

Although the concept of *mixins* originated in work on Common LISP from the mid-1980s [25], Bracha and Cook [4] were the first to propose mixins as an actual language construct (in their case, as an extension to Modula-3). Since then, mixins have appeared in a variety of different languages, under a variety of different names, meaning a variety of different (albeit related) things.

In the context of Bracha and Cook's pioneering work, as well as most subsequent object-oriented instances of mixins, a mixin is an *abstract subclass* (their terminology), *i.e.,* a subclass that is parameterized over an abstract specification of its superclass and can be instantiated to extend multiple different superclasses.

The other common meaning of mixins, which is less specific to object-oriented programming and is the one we are primarily interested in for the purposes of this paper, is also due to Bracha, in particular his work with Lindstrom on the Jigsaw language [5]. Jigsaw's central construct is actually not called a mixin, but rather a *module*. Jigsaw modules may contain both *defined* components (*i.e.,* exports) and *declared* components (*i.e.,* imports). The language provides a suite of operators for adapting and combining modules. Of particular note is the *merge* operator, which takes as input two modules, M_1 and M_2, and returns a module M such that

1. $exports(\mathrm{M}) = exports(\mathrm{M}_1) \uplus exports(\mathrm{M}_2)$
2. $imports(\mathrm{M}) = imports(\mathrm{M}_1) \cup imports(\mathrm{M}_2) - exports(\mathrm{M})$

Here, $\uplus$ denotes that the exports of M_1 and M_2 must be disjoint. In addition, the typing rule for the merge operator checks that any components with the same name in M_1 and M_2 have compatible types (for some suitable definition of "compatible", *e.g.,* the types are equal, or one is a subtype of the other).

While the merge operator does not permit M_1 and M_2 to have overlapping exports, Bracha provides a separate *override* operator that does, choosing the export from M_2 over the export from M_1 in case of an overlap. In some later versions of mixins, a variant of the override operator, not the merge operator, is adopted as the default notion of mixin composition. Moreover, support for overriding (and "late binding") is often considered a central feature of mixins. Be that as it may, for the remainder of this paper we will use the term *mixin composition* to mean Bracha's merge operator. Following mixin-based languages like Flatt *et al.*'s *units* [14, 30] and Duggan's *recursive DLLs* [12], our MixML language does not attempt to support any form of overriding.

The work on Jigsaw has inspired a significant amount of research into *mixin module systems*. Over the course of several papers [3, 2, 1], Ancona and Zucca have explored in depth the semantic properties and algebraic laws of mixin operators, and developed a foundational mixin module calculus called CMS, which refactors some of the Jigsaw primitives. While CMS is a pure call-by-name language, it has been extended with support for call-by-value evaluation [17] and monadic effects [1].

Compared with ML modules, a key advantage of mixin modules is that the mixin composition of modules M_1 and M_2 is by definition a kind of recursive linking, in which the exports of each module are used to satisfy the imports of the other. Mixin modules thus appear to offer a natural solution to the problems with separate compilation of recursive modules in ML. One major limitation of Bracha/CMS-style modules, however, is that they contain only term components, not type components, which means that they cannot express ML-style abstract data types, let alone translucent signatures. This has led a number of researchers to consider ways of combining the support for type abstraction found in ML modules with the support for separate compilation and recursive linking found in mixin modules [14, 12, 29, 30].

1.3 Motivation

The motivation for this paper is that the existing proposals for synthesizing ideas from ML modules and mixin modules (which we will discuss in detail in Section 6) are all lacking in one key respect: none of them allows for a simple and direct encoding of all the salient, semantically interesting features of the ML module system. For example, Owens and Flatt [30] give an encoding of an ML-like module calculus into their *unit* language, but it depends critically on the impractical assumption that the ML programs being encoded have (redundant) signature annotations on nearly every subexpression. Odersky *et al.*'s Scala language [29], while highly expressive in its support for OO-style extensibility, does not support opaque signature ascription—*i.e.,* the ability to seal a module (or class) *ex post facto* behind an abstract interface—which is a central feature of the ML module system. Ideally, we would like a language that seamlessly integrates mixin composition into ML *without* sacrificing any key features of ML modules.

1.4 MixML

In this paper, we present a novel foundational module system, called MixML, which incorporates at a deep level the mechanism of mixin module composition, while retaining the full expressive power of ML modules.

The main idea of MixML is simple: the MixML module language unifies ML's structure and signature languages into one. That is, a MixML module may contain both type and term *definitions*, of the kind found in ML structures, as well as type and term *specifications*, of the kind found in ML signatures. It is not required to contain only definitions or only specifications; rather, it may freely mix them. Thus, traditional ML structures and ML signatures may be viewed as endpoints on the spectrum of MixML modules.

Why is MixML's unification of structures and signatures useful? Because it enables us to encode a wide variety of features directly as stylized uses of a small set of orthogonal underlying constructs, thus simplifying and regularizing the design of the language. In particular:

1. MixML provides a unifying account of several pairs of language features that are usually modeled as extensions to *both* the structure and signature languages of ML. Concretely, for each of the following pairs of features, MixML supports both features via a single encoding:
   - hierarchical structures and hierarchical signatures
   - recursive structures and recursively dependent signatures
   - functors and parameterized signatures
2. A variety of features that are typically supported via distinct mechanisms may be encoded in MixML as idiomatic uses of mixin composition, *i.e.,* (recursive) linking. These include:
   - recursive module definitions
   - signature inheritance (`include`)
   - signature refinement (`with`/`where`/`sharing type`)
   - signature ascription, opaque (`:>`) and transparent (`:`)
   - functor application

   The encodings of these features involve the linking of two MixML modules, one or both of which represents an ML signature. These encodings are made possible by the fact that structures and signatures are just different kinds of MixML modules, and any two modules can be linked together so long as they are *compatible* (in a sense we make precise later in the paper).

1.5 Technical Contributions

If the basic design idea of MixML is as simple and powerful as we claim, the reader may wonder why it has not been proposed before. We believe the reason is that the feasibility of the idea is dependent on several novel enhancements to mixin module semantics that we set forth in this paper, as well as a generalization of some recent work on handling the "double vision" problem in the context of recursive ML-style modules. We briefly summarize these technical contributions here.

Hierarchical Composability Suppose `M1` and `M2` are ML structures, each of signature

```
sig val x : int; val y : int end
```

One can compose them hierarchically to form a new structure containing both:

```
module M = struct
  module A1 = M1; module A2 = M2
end
```

If `M1` and `M2` were *mixin* modules, each with `x` as an import and `y` as an export, we might wish to hierarchically compose them in the same way, with the result being a new mixin module `M`, with imports `A1.x` and `A2.x`, and exports `A1.y` and `A2.y`. Yet, in previous CMS-style mixin module systems, hierarchically composing two mixins to form another mixin is not possible. The reason is that CMS-style systems employ a flat namespace for their imports and exports—pathnames like `A1.x` are disallowed.

Hierarchical composability, which MixML modules support, allows us to use a single *namespace* mechanism to build hierarchies of structures, signatures, or modules that are a mixture of both. Without it, we would be unable to provide a unified representation of hierarchical structures and signatures.

Unifying Linking and Binding In previous systems, the mixin composition of two modules does not provide a way for either of the modules to refer directly to components of the other. In other words, the linking operator is not a variable binder; instead, binding is typically built into other constructs.

In MixML, we take a different tack by making the linking operator the *only* binding construct. This enables us to (1) model all forms of binding in ML modules as stylized uses of linking, and (2) achieve very simple encodings of several features, such as recursive modules and `sharing type` specifications. The benefit of unifying linking and binding will be borne out by a number of examples in Section 2.

Cross-Eyed Double Vision A key problem that arises when extending ML with recursive modules is *double vision* [6, 7], which concerns the interaction of recursion and opaque signature ascription. As MixML modules subsume the functionality of recursive ML modules, double vision is an issue for MixML as well. In fact, since mixin composition is essentially a bidirectional generalization of ML-style signature matching, the MixML type system must handle a "cross-eyed" version of the double vision problem.

Fortunately, in recent work, Dreyer [7, 9] has shown how to solve the double vision problem for recursive ML modules, and this solution can be generalized quite easily to handle the cross-eyed double vision problem for MixML modules. We describe the problem and its solution by example in Section 3, and provide full formal details of the solution in Section 4.

1.6 Overview

The rest of the paper is structured as follows. In Section 2, we present the syntax of MixML and lead the reader on a tour of the language by example. In Section 3, we explore several technical issues that the MixML type system must address, including the double vision problem and the handling of cyclic definitions. In Section 4, we give the formal definition of the MixML type system, in particular the static semantics. (Due to space limitations, we omit most details concerning the dynamic semantics and type soundness proof, and refer the interested reader to the expanded online report [11].) In Section 5, we describe an extension of MixML with support for higher-order modules. Finally, in Section 6, we offer a detailed comparision with related work, and conclude with directions for future work.

2. A Tour of MixML

The syntax of MixML is displayed in Figure 1.

In a MixML module, some components may be defined (the exports), and some may have a kind or type specification but are not defined (the imports). The import components of a module can be viewed as requirements that will be fulfilled in the future when the module is linked with other modules. Thus, the MixML type system insists that no module operator be permitted to remove the imports of a module from scope (*e.g.,* by the use of data abstraction), as one should not be allowed to forget about a requirement. In contrast, exports may always be hidden.

Types and Terms Following Leroy [21], we define our module language to be largely agnostic with respect to the details of the core language. Of the term language we expect only that it contains a *term projection* construct $\texttt{Val}(\mathit{mod})$, which takes an atomic term module mod (*i.e.,* a module containing a single term component) and projects out the term. Similarly, we assume that the type language contains a *type projection* construct $\texttt{Tyc}(\mathit{mod})$, which takes an atomic type module mod (*i.e.,* a module containing a single type component) and projects out the type. For brevity, we will typically omit the explicit `Tyc` and `Val` projections in examples when their necessity is clear from context.

We also assume that the type language contains ML-style type constructors, which take a list of type arguments (the notation $\overline{\alpha}$ denotes a list of 0 or more α's, separated by commas) and return a single type as a result. The kind of a type constructor is its arity, *i.e.,* the number of type arguments it has. Types classifying terms have kind 0.

Atomic Modules Atomic modules are modules containing a single (type or term) component, and that component may be either specified (*i.e.,* an import) or defined (*i.e.,* an export). Whereas, in ML, definitions only occur in modules, and specifications only occur in signatures, in MixML both definitions and specifications are module constructs.

The module $\texttt{[:}\mathit{tyc}\texttt{]}$ represents a term specification of type tyc. The module $\texttt{[}\mathit{exp}\texttt{]}$ represents a term component defined to be the value resulting from the evaluation of exp. The module $\texttt{[:K]}$ represents an abstract type specification of kind K. The module $\texttt{[}\mathit{tyc}\texttt{]}$ represents a transparent definition of a type component equal to tyc. Note that, in ML, there is a distinction between transparent type *definitions*, which appear in modules, and transparent type *specifications*, which appear in signatures. In MixML, these mechanisms are unified into one.

Unary Namespaces and Projection The construct $\{\}$ denotes an empty module, containing no components. The module $\{\ell = \mathit{mod}\}$ introduces a namespace containing a single component named ℓ, whose definition is mod. Any imports (resp. exports) of mod become imports (resp. exports) of $\{\ell = \mathit{mod}\}$ as well, except the pathnames of those imports (resp. exports) now have "ℓ." in front of them. Thus, MixML modules are *hierarchically composable*.

The constructs we have discussed so far can be combined to give a direct encoding of ML-style type/term definitions/specifications:

$$\begin{array}{lcl}
\texttt{val v : } \mathit{tyc} & \stackrel{\text{def}}{=} & \{\texttt{v} = \texttt{[:}\mathit{tyc}\texttt{]}\} \\
\texttt{val v = } \mathit{exp} & \stackrel{\text{def}}{=} & \{\texttt{v} = \texttt{[}\mathit{exp}\texttt{]}\} \\
\texttt{type (}\alpha_1,\ldots,\alpha_n\texttt{) t} & \stackrel{\text{def}}{=} & \{\texttt{t} = \texttt{[:}n\texttt{]}\} \\
\texttt{type (}\overline{\alpha}\texttt{) t = } \mathit{tyc} & \stackrel{\text{def}}{=} & \{\texttt{t} = \texttt{[}\lambda(\overline{\alpha}).\mathit{tyc}\texttt{]}\}
\end{array}$$

Dual to $\{\ell = \mathit{mod}\}$ is the construct $\mathit{mod}.\ell$, which projects the ℓ component from the module mod. The typing rule for $\mathit{mod}.\ell$ insists that any imports mod may have must be contained in the ℓ component. This guarantees that no import requirements of mod are dropped when we project out the ℓ component.

$$
\begin{array}{lrcl}
\text{Kinds} & \mathrm{K} & \in & \mathbb{N} \\
\text{Type Var's} & \alpha, \beta & \in & \mathit{TyVars} \times \mathbb{N} \\
\text{Module Var's} & \mathrm{X}, \mathrm{Y} & \in & \mathit{ModVars} \\
\text{Labels} & \ell & \in & \mathit{Labs} \\
\text{Label Paths} & \ell s & ::= & \epsilon \mid \ell.\ell s \in \mathit{Paths} \\
\text{Type Constr's} & \mathit{tyc} & ::= & \mathtt{Tyc}(\mathit{mod}) \mid \\
 & & & \alpha \mid \lambda(\overline{\alpha}).\mathit{tyc} \mid \mathit{tyc}(\overline{\mathit{tyc}}) \mid \ldots \\
\text{Terms} & \mathit{exp} & ::= & \mathtt{Val}(\mathit{mod}) \mid \ldots \\
\text{Modules} & \mathit{mod} & ::= & \mathrm{X} \mid \{\} \mid \mathtt{[:}\mathit{tyc}\mathtt{]} \mid \mathtt{[}\mathit{exp}\mathtt{]} \mid \\
 & & & \mathtt{[:}\mathrm{K}\mathtt{]} \mid \mathtt{[}\mathit{tyc}\mathtt{]} \mid \\
 & & & \{\ell = \mathit{mod}\} \mid \mathit{mod}.\ell \mid \\
 & & & (\mathrm{X} = \mathit{mod}_1)\ \mathtt{with}\ \mathit{mod}_2 \mid \\
 & & & (\mathrm{X} = \mathit{mod}_1)\ \mathtt{seals}\ \mathit{mod}_2 \mid \\
 & & & \mathtt{[}\mathit{mod}\mathtt{]} \mid \mathtt{new}\ \mathit{mod}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{mod}_1\ \mathtt{with}\ \mathit{mod}_2 & \stackrel{\text{def}}{=} & (\mathrm{X} = \mathit{mod}_1)\ \mathtt{with}\ \mathit{mod}_2 \\
\mathit{mod}_1\ \mathtt{seals}\ \mathit{mod}_2 & \stackrel{\text{def}}{=} & (\mathrm{X} = \mathit{mod}_1)\ \mathtt{seals}\ \mathit{mod}_2
\end{array}
$$

$$\text{where } \mathrm{X} \notin \mathsf{FV}(\mathit{mod}_2)$$

Figure 1. MixML Syntax

At the moment, all we have are unary namespaces. In order to support n-ary structures and signatures of the sort found in ML, we now present MixML's most versatile construct—*linking*.

Linking The linking module construct $(\mathrm{X} = \mathit{mod}_1)$ `with` mod_2 is MixML's primary means of composing multiple modules together. Linking does several things:

- It performs mixin composition of mod_1 and mod_2 in the style of Bracha's merge operator, assuming they are compatible.
- It sequences effects. Any definitions of term components in mod_1 will be evaluated prior to any such definitions in mod_2.
- It is the only means of variable binding in the language. It binds X as a representative of mod_1 inside mod_2.

Compatibility of mod_1 and mod_2 reduces to compatibility of their atomic components. For each component of the same pathname in both modules, compatibility is defined informally as follows:

- If the component is an import in mod_1 and an export in mod_2, then mod_2's export must match the import spec from mod_1. (And vice versa, if the component is an export in mod_1 and an import in mod_2.)
- If the component is a term import in both modules, and has specification tyc_1 in mod_1 and tyc_2 in mod_2, then either tyc_1 must be a subtype of tyc_2 (for some notion of core subtyping, *e.g.,* polymorphic instantiation) or vice versa, and whichever type is stronger is the one propagated as the specification of the component in the linked module.
- If the component is a type import in both modules, they must both specify it to have the same kind.
- If the component is a type export in both modules, they must both define it to be the same type.
- The component may not be a term export in both modules.

Before exploring the recursive aspects of linking, we first show how it may be used to express n-ary non-recursive structures and signatures, as well as several other non-recursive features.

n-ary Structures and Signatures While, in ML, components of a structure or signature are for convenience only assigned a single name, most type-theoretic accounts of the ML module system employ a *label-variable distinction* [15] (or the equivalent [20]). This divides the name of each component into a *label* ℓ, which is unchangeable and is used as the "external" name of the component, and a *variable* X, which is alpha-convertible and is used as the "internal" name of the component within subsequent definitions/specifications of the structure/signature. Under this approach, an n-ary structure can be modeled as

$$\{\ell_1 \triangleright \mathrm{X}_1 = \mathit{mod}_1, \ldots, \ell_n \triangleright \mathrm{X}_n = \mathit{mod}_n\}$$

where each X_i is bound in the subsequent mod_j's to the result of evaluating mod_i. (In ML, ℓ_i and X_i must be the same identifier.)

The encoding of an n-ary structure defines it as the linking of n disjoint unary structures (we assume here that X is suitably fresh):

$$\{\ell_1 \triangleright \mathrm{X}_1 = \mathit{mod}_1, \ldots\} \stackrel{\text{def}}{=} (\mathrm{X} = \{\ell_1 = \mathit{mod}_1\})\ \mathtt{with}\ \{\ldots\}[\mathrm{X}.\ell_1/\mathrm{X}_1]$$

Inside the linking, X stands for the unary structure containing just the ℓ_1 component. In the encoding of the remainder of the components $(\ldots)$, we must therefore replace references to X_1 with $\mathrm{X}.\ell_1$. (Note that we assume here for simplicity that all the ℓ_i are distinct labels. There are well-known ways of allowing for shadowing [16], but they are beyond the scope of this paper.)

Typically, ML module type systems model n-ary signatures in a similar fashion to n-ary structures (yet as a distinct construct):

$$\{\ell_1 \triangleright \mathrm{X}_1 : \mathit{sig}_1, \ldots, \ell_n \triangleright \mathrm{X}_n : \mathit{sig}_n\}$$

However, since ML signatures are encoded in MixML as modules (*i.e.,* a *sig* is just a module with no term exports), the encoding of n-ary signatures is exactly the same as for n-ary structures:

$$\{\ell_1 \triangleright \mathrm{X}_1 : \mathit{sig}_1, \ldots\} \stackrel{\text{def}}{=} (\mathrm{X} = \{\ell_1 = \mathit{sig}_1\})\ \mathtt{with}\ \{\ldots\}[\mathrm{X}.\ell_1/\mathrm{X}_1]$$

We just *change the colons to equal signs*: wherever you see $\mathrm{X} : \mathit{sig}$ in ML code, expect to see $\mathrm{X} = \mathit{sig}$ in its MixML encoding. As we will show, this maxim applies to all instances of structure specification in ML, not just substructure specifications.

Local Module Definitions Using the above encoding of n-ary namespaces, we can easily encode a `let` construct that enables the definition of *local* modules. The encoding makes use of two labels, ℓ_1 and ℓ_2, which are arbitrary:

$$\mathtt{let}\ \mathrm{X} = \mathit{mod}_1\ \mathtt{in}\ \mathit{mod}_2 \stackrel{\text{def}}{=} \{\ell_1 \triangleright \mathrm{X} = \mathit{mod}_1, \ell_2 = \mathit{mod}_2\}.\ell_2$$

The two modules mod_1 and mod_2 are combined through hierarchical composition into a pair module, from which the second component ℓ_2 is then projected out. Since this has the effect of hiding mod_1, the MixML type system will insist that mod_1 be *complete*, *i.e.,* that it have no imports. This is a useful property to be able to enforce. Thus, in general, if we wish to check that a module *mod* is complete, we can do so by just `let`-expanding it:

$$\mathtt{let}\ \mathrm{X} = \mathit{mod}\ \mathtt{in}\ \mathrm{X}$$

Signature Inheritance In ML, one may define a signature that inherits specifications from an existing signature *sig* and adds new specifications to it. This is supported by the `include` mechanism:

`sig include` *sig*`; <`*newspecs*`> end`

MixML supports signature inheritance through linking. To add `<`*newspecs*`>` to *sig*, we can write

$(\mathrm{X} = \mathit{sig})$ `with` $\{$`<`*newspecs*`>`$\}$

(Note that in our encoding, in order for `<`*newspecs*`>` to refer to the components specified in *sig*, it must project them from X.)

In fact, linking is more flexible than `include` because `include` does not permit multiple inheritance from overlapping signatures. For instance, if sig_1 and sig_2 both contain specifications of a type component (named `t` in both signatures), together with several operations over values of that type, it is prohibited to write

$$\texttt{sig include } sig_1\texttt{; include } sig_2 \texttt{ end}$$

due to the overlapping `t` specs. With linking, though, we can write

$$sig_1 \texttt{ with } sig_2$$

and mixin composition will permit overlapping specs in sig_1 and sig_2 so long as they are compatible (which in this case they are). A similar approach to multiple signature inheritance is offered by Ramsey *et al.*'s `andalso` signature combinator [31], but in our case the added functionality falls out directly from the semantics of mixin linking.

ML also provides an `open` mechanism for *structure* inheritance. If `open` were a non-shadowing operation like signature `include`, the encoding of `include` would double as an encoding of `open` (replacing the *sig*'s above with *mod*'s). However, unlike signature inclusion, `open` is permitted to shadow earlier bindings. As mentioned above, we believe it should be straightforward to extend MixML with support for shadowing using known techniques [16].

Signature Refinement Mixin linking can also be used to define a very simple encoding of ML's `with type` (or `where type`) mechanism for adding type definitions to signatures. The ML construct

$$sig \texttt{ with type (}\overline{\alpha}\texttt{) t = } tyc$$

can be modeled (quite directly!) as a form of linking:

$$sig \texttt{ with type (}\overline{\alpha}\texttt{) t = } tyc$$

where "`type` $(\overline{\alpha})$ `t =` *tyc*" is encoded as on the previous page. Mixin linking will use the definition for `t` on the right side of the `with` (*i.e.,* $\lambda(\overline{\alpha}).tyc$) to fill in the abstract specification for `t` in *sig*. It is also easy to encode the more general form of `with type` in which `t` can be a path $\ell_1.\cdots.\ell_n$, using the following definition:

$$\texttt{type (}\overline{\alpha}\texttt{)}\ \ell_1.\ell s \texttt{ = } tyc \stackrel{\text{def}}{=} \{\ell_1 = (\texttt{type (}\overline{\alpha}\texttt{)}\ \ell s \texttt{ = } tyc)\}$$

In fact, though, the above encoding is not entirely faithful to the original ML semantics of `with type`: if the type component `t` does not appear in *sig* at all, then the encoding will not report a type error (as ML semantics would dictate it should), but rather simply add "`type` $(\overline{\alpha})$ `t =` *tyc*" to *sig*. If we want to match ML semantics more precisely, we need to first check that *sig* contains a specification for the type `t`. This can be achieved by replacing *sig* in the above encoding with *sig* `matches type` $(\overline{\alpha})$ `t`, where the `matches` mechanism is encoded as follows:

$$sig_1 \texttt{ matches } sig_2 \stackrel{\text{def}}{=}$$
$$\{\ell_1 \rhd \mathrm{X} = sig_1, \ell_2 = \mathrm{X} \texttt{ with } sig_2\}.\ell_1$$

The projection of ℓ_1 here means that: (1) if the encoding is well-typed, then sig_1 `matches` sig_2 is indistinguishable from sig_1, and (2) the encoding will only be well-typed if the hidden module labeled ℓ_2 is complete (no imports). This second condition implies that the imports of sig_2 (*i.e.,* its value and abstract type specs) must be satisfied by linking with X—*i.e.,* X's signature sig_1 must actually *match* sig_2.

Type Sharing Constraints If *sig* contains two abstract type components `u` and `t`, and we wish to refine the signature so that `t` is transparently equal to `u`, the traditional ML `with type` construct does not permit us to do so because `u` is not a valid type outside the signature. Standard ML retains a second signature refinement operator, `sharing type`, precisely to make up for this deficiency.

In MixML, we can encode `sharing type` very easily by exploiting the ability to bind *sig* to a variable while we refine it. That is, in order to refine *sig* so that `t` equals `u`, we can write

$$(\mathrm{X} = sig) \texttt{ with type t = X.u}$$

We use X here to provide `t`'s definition with a way of referring to the `u` component from *sig*. This is similar to a proposal of Ramsey *et al.* [31], but in our case the added functionality falls out directly from our unification of linking and binding.

Recursive Structures Another feature for which the unification of linking and binding facilitates a very simple encoding is the recursive structure definition. Recursive structure extensions to ML typically have the form:

$$\texttt{rec}\,(\mathrm{X} : sig)\ mod$$

Here, X is the variable by which *mod* refers to itself recursively, and *sig* is the *forward declaration*, a kind of template for *mod*, which serves as the signature of X during the typechecking of *mod*.

The encoding of this construct in MixML is extremely simple:

$$(\mathrm{X} = sig) \texttt{ with } mod$$

Mixin linking will use the type definitions (type exports) of *mod* to fill in the corresponding abstract type specifications (type imports) of *sig*, and then check that the term definitions (term exports) of *mod* match the types from the corresponding term specifications (term imports) of *sig*. The binding of X inside *mod* gives *mod* a way of referring to its own components (at least those specified in *sig*) recursively. Note also that this is another instance of our rule: Just *change the colons to equal signs*—X : *sig* becomes X = *sig*.

One thing this encoding will not do in its present form is ensure that all the components forward-declared in *sig* actually get defined by *mod*. Any components that *mod* fails to define will just remain imports in the result of the linking. To ensure completeness, though, we can simply `let`-expand the encoding, as explained above.

Recursively Dependent Signatures All the existing recursive module proposals for ML also extend the signature language with a new construct called a *recursively dependent signature* [6]. In Russo's extension to Moscow ML [33], it takes the form:

$$\texttt{rec}\,(\mathrm{X} : sig_1)\ sig_2$$

This construct allows the signatures of mutually recursive modules (in sig_2) to refer recursively to each other's type components through the variable X. Of course, since structures and signatures are both encoded in MixML as modules, this construct is encoded in the exact same way as the recursive structure construct:

$$(\mathrm{X} = sig_1) \texttt{ with } sig_2$$

One point of note is that not all recursive module extensions to ML require the programmer to write down sig_1. Instead, they infer it from sig_2. We view such an inference step as a separable convenience. In any case, this encoding demonstrates that recursive structures and recursively dependent signatures can be understood as one and the same feature.

Opaque Signature Ascription as Opaque Linking None of the MixML constructs described thus far supports the creation of abstract data types. For this purpose MixML includes a second variant of the linking construct—$(\mathrm{X} = mod_1)$ `seals` mod_2—which we call *opaque linking* (as opposed to the original form, which we view as *transparent linking*).

Opaque linking is very similar to transparent linking, *except*:

- The only information that the rest of the program may know about the result of opaque linking is what it can tell from looking at mod_1—no information about mod_2 may be revealed.

This property implies several things. First, mod_2 must define *all* of mod_1's imports. If it only defined some of them, we would have no way of knowing which ones it defined without looking at it (and thus violating data abstraction). Second, the type imports of mod_1 will become *abstract type exports* of the linked module. Third, the exports of the linked module are limited to those components either specified (imports) or defined (exports) in mod_1. Thus, since all of mod_1's imports are fulfilled by mod_2, the result of opaque linking is always a complete module.

Using opaque linking, we arrive at a simple encoding of ML's sealing (or opaque signature ascription) construct. Specifically:

$$mod :> sig \overset{\text{def}}{=} sig \; \texttt{seals} \; mod$$

Functors as Units So far we have not introduced any means of *suspending* a module in the manner of an ML functor. To support this important feature, we introduce a new atomic module construct we call a *unit*. (As we explain in Section 6, our units are inspired by Flatt *et al.*'s units [14, 30], but are different in many respects.)

A unit, written `[`mod`]`, is a suspension of the module mod. With units we can encode an ML functor (modeled here by a module-level λ-expression) as follows:

$$\lambda(\mathrm{X}{:}sig).mod \overset{\text{def}}{=} \texttt{[}\{\texttt{Arg} \triangleright \mathrm{X} = sig,\ \texttt{Res} = mod\}\texttt{]}$$

In other words, a functor is just a suspension of a module with one component `Arg` whose term (and possibly type) components are undefined, and one component `Res` that is fully defined. (Note how the argument binding $\mathrm{X} : sig$ is encoded as $\mathrm{X} = sig$, yet another application of the rule of changing colons to equal signs.)

The elimination construct for units is written `new` mod. Here, mod is assumed to be a unit, and `new` mod has the effect of *instantiating* that unit by producing a fresh copy of its constituent module, which can then be linked with other modules that satisfy its imports. For example, suppose that the variable F has been bound to the functor expression shown above. Application of F to an argument mod is encoded as follows:

$$\mathrm{F}(mod) \overset{\text{def}}{=} (\{\texttt{Arg} = mod\} \; \texttt{with} \; \texttt{new} \; \mathrm{F}).\texttt{Res}$$

The reason we put `new` F on the r.h.s. of the linking is to ensure that the term definitions in mod are evaluated *before* the term definitions in the body of F, which may depend on them.

Every instantiation of a unit F generates a distinct instance of the module expression contained within F. In particular, each occurrence of `new` F will re-evaluate the term definitions in F's constituent module and generate fresh abstract types corresponding to said module's abstract type exports. In this respect, unit instantiation is much like generative functor application in Standard ML. We do not currently model the *applicative* behavior of functors in OCaml [19], which we leave to future work.

Transparent Signature Ascription In addition to opaque signature ascription, Standard ML includes a mechanism for transparent signature ascription, written $mod : sig$, which narrows the exports of mod to those specified in sig but does not perform any type abstraction. It is well-known that transparent ascription with signature sig can be encoded as an application of the identity functor at sig:

$$mod : sig \overset{\text{def}}{=} (\lambda(\mathrm{X}{:}sig).\mathrm{X})(mod)$$

Parameterized Signatures Jones [18] proposed the idea of parameterized signatures, *i.e.,* signatures parameterized over module arguments. Although it has been argued that ordinary ML signatures subsume the expressiveness of parameterized signatures, we merely wish to point out here that parameterized signatures are directly encodable in MixML via the *exact same encoding* as for functors. Just instantiate mod in $\lambda\mathrm{X}{:}sig.mod$ with the MixML representation of an ML signature. Functors and parameterized signatures are one and the same thing.

Signature Bindings In addition to modeling parameterized structures/signatures, units can be used to model ML's `signature` bindings. Suppose sig is an ML signature encoded as a MixML module, and that we wish to bind it to a *signature variable* S (to use as shorthand for sig in subsequent code). It would be incorrect to bind S to sig directly: $\mathrm{S} = sig$ is the MixML encoding of the ML structure specification $\mathrm{S} : sig$, so if S were defined that way, later references to it would be references to a particular structure of signature sig. In order to define S to *be* the signature sig, we bind S to the *unit* `[`sig`]` and replace all subsequent uses of S with `new` S. This works because each reference to `new` S will produce a fresh copy of sig, whose imports may then be instantiated independently. Since ML signatures do not contain term definitions, performing `new` on a signature variable will never have any computational effects.

Separate Compilation of Recursive Modules At the start of the paper, the main criticism we gave of ML modules was that they do not support separate compilation of mutually recursive modules. In MixML, this functionality is provided by units.

Suppose we wish to define two modules named `A` and `B`, with signatures $sig_{\texttt{A}}$ and $sig_{\texttt{B}}$, and definitions $mod_{\texttt{A}}$ and $mod_{\texttt{B}}$, which refer recursively to themselves and to each other through the module variable X. Let the signature variable S be bound to the unit

$$\texttt{[}(\mathrm{X} = \ldots) \; \texttt{with} \; \{\texttt{A} = sig_{\texttt{A}}, \texttt{B} = sig_{\texttt{B}}\}\texttt{]}$$

where ... is a signature specifying the type components of `A` and `B` that $sig_{\texttt{A}}$ and $sig_{\texttt{B}}$ need to refer to recursively. Were we to write `A`'s and `B`'s definitions together, the MixML code would be:

$$(\mathrm{X} = \texttt{new}\ \mathrm{S}) \; \texttt{with} \; \{\texttt{A} = mod_{\texttt{A}}, \texttt{B} = mod_{\texttt{B}}\}$$

But there is no need to define `A` and `B` together. We can, separately, bind $\texttt{U}_{\texttt{A}}$ to

$$\texttt{[}(\mathrm{X} = \texttt{new}\ \mathrm{S}) \; \texttt{with} \; \{\texttt{A} = mod_{\texttt{A}}\}\texttt{]}$$

and $\texttt{U}_{\texttt{B}}$ to

$$\texttt{[}(\mathrm{X} = \texttt{new}\ \mathrm{S}) \; \texttt{with} \; \{\texttt{B} = mod_{\texttt{B}}\}\texttt{]}$$

The units $\texttt{U}_{\texttt{A}}$ and $\texttt{U}_{\texttt{B}}$ represent the separately compiled versions of `A` and `B`, respectively. $\texttt{U}_{\texttt{A}}$ exports definitions for the components of `A`, but leaves `B`'s components as imports, and $\texttt{U}_{\texttt{B}}$ is vice versa. Finally, when we want to link them we simply write:

$$\texttt{new}\ \texttt{U}_{\texttt{A}} \; \texttt{with} \; \texttt{new}\ \texttt{U}_{\texttt{B}}$$

Of course, there is nothing requiring us to link $\texttt{U}_{\texttt{A}}$ and $\texttt{U}_{\texttt{B}}$ in this order or with each other. They are completely independent program units that can link with any other compatible units.

Higher-Order Units With the constructs presented so far, units can only be defined and *exported* from other units. If we wish to support the expressiveness of higher-order functors (functors that take functors as arguments—a feature in many dialects of ML), then the language must also support unit *imports*. In order to encode unit imports, it is necessary to extend MixML with a notion of *unit signature*, analogous to the notion of *functor signature* in higher-order module extensions to ML.

As this extension is completely orthogonal to the other features of MixML, and since we feel the rest of the MixML type system is perhaps easier to understand in the absence of this extension, we will continue in the next two sections by presenting the basic MixML type system, and then present higher-order units and unit signatures as an optional extension in Section 5.

3. Challenges in Typing MixML

The combination of recursive mixin composition with abstract type components and sealing raises a number of technical challenges. In this section we informally discuss the central problems that arise in typing MixML. The solutions we employ are mostly generalizations of the techniques developed by Dreyer for typing recursive modules [7].

Bidirectional Type Lookup In ML, matching a structure mod against a signature *sig* is a two-step procedure. First, for each type component specified abstractly in *sig*, we look up its definition in mod, and *reify* its specification in *sig* appropriately (*i.e.,* make its specification transparently equal to its definition in mod). We then check that the specification of each (type or value) component in the reified *sig* is matched by its corresponding definition in mod.

To see why this two-step process is necessary, consider:

```
struct type t = int; val x = 3 end
                 :>
      sig type t; val x : t end
```

Checking whether 3 has type `t` will fail unless we first reify the specification of `t` to its underlying definition, `type t = int`.

In MixML, we no longer explicitly distinguish structures from signatures. Linking effectively generalizes unidirectional matching to bidirectional *merging* of two modules, which may both contain abstract type components. Consequently, reification (type lookup) must be performed in both directions simultaneously. Consider:

$$\left\{\begin{array}{lcl} \texttt{t}\triangleright\texttt{t}_1 & = & \texttt{[int]},\\ \texttt{u}\triangleright\texttt{u}_1 & = & \texttt{[:0]},\\ \texttt{f} & = & \texttt{[:int}\rightarrow\texttt{u}_1\texttt{]}\end{array}\right\} \;\texttt{with}\; \left\{\begin{array}{lcl} \texttt{t}\triangleright\texttt{t}_2 & = & \texttt{[:0]},\\ \texttt{u}\triangleright\texttt{u}_2 & = & \texttt{[bool]},\\ \texttt{f} & = & \texttt{[}\lambda x\texttt{:t}_2\texttt{.true]}\end{array}\right\}$$

The definition for `f` in the second module will only match its specification in the first module if we first reify `u` to `bool` and `t` to `int` in both modules. This involves *bidirectional type lookup*, a straightforward generalization of ML's unidirectional type lookup.

Cyclic Type Definitions Type lookup in MixML can easily introduce cyclic type definitions. For example,

$$(\texttt{X} = \{\texttt{t} = \texttt{[:0]}\})\;\texttt{with}\;\{\texttt{t} = \texttt{[X.t}\rightarrow\texttt{int]}\}$$

or

$$\left\{\begin{array}{lcl} \texttt{t}\triangleright\texttt{t} & = & \texttt{[:0]},\\ \texttt{u}\triangleright\texttt{u} & = & \texttt{[t]}\end{array}\right\} \;\texttt{with}\; \left\{\begin{array}{lcl} \texttt{u}\triangleright\texttt{u} & = & \texttt{[:0]},\\ \texttt{t}\triangleright\texttt{t} & = & \texttt{[u]}\end{array}\right\}$$

Supporting such definitions would require the introduction of higher-kinded *equi-recursive* types [6] into the type system, for which there is no known effective typechecking algorithm in the general case. Hence, during type lookup, we check that the definitions of the abstract type components being looked up do not have cyclic dependencies. In particular, we would reject both of the above examples.

The prohibition on type cycles during lookup prevents one from defining *transparently* recursive types. However, as we explain at the end of this section, we do allow the definition of *opaquely* recursive types, which generalize ML-style iso-recursive datatypes.

Cyclic Term Definitions Linking can also introduce cycles between the definitions of term components. We adopt a call-by-value semantics for the evaluation of term components in MixML, where recursion is implemented by `letrec`-style backpatching. This is similar to the approach taken by several other recursive module systems [14, 33, 22, 26, 7].

Under the backpatching semantics, cyclic linking can cause a run time exception if a term component is accessed before its definition has been evaluated. Static detection of such errors is a problem that is orthogonal to our work and has been addressed by Hirschowitz and Leroy [17] and Dreyer [8] among others.

Double Vision An important problem that arises in extending ML with recursive modules is the *double vision problem* [7]. Consider the following simple example:

$$(\texttt{X} = \{\texttt{t} = \texttt{[:0]}, \ldots\})\;\texttt{seals}\;\{\texttt{t} = \texttt{[int]}, \ldots\}$$

Here, we are defining a recursive sealed module with a type component `t` that is defined internally to be `int`. Within the r.h.s. of the `seals`, we know that `t` is implemented as `int`, so we ought to know that `X.t` (which is just a recursive reference to `t`) equals `int` as well, but the signature bound to `X` does not reflect this. As a result, the programmer may be forced to expose the definition of `t` as `int` in the l.h.s. module, thus losing type abstraction.

For this particular example, the problem can be worked around by making `t` transparently equal to `int` in the l.h.s. module, and then applying sealing "after the fact." However, it is not always possible to seal after the fact. For instance, if a recursive module contains sealed substructures that wish to hide type information from one another, then there is no way to hoist out the sealing without exposing the substructures' implementations to each other.

Fortunately, Dreyer has recently developed a general solution to the double vision problem in his RMC type system [7], and we can readily adopt his solution. The central ideas of RMC are as follows.

First is the idea of *forward-declaring* abstract types. In RMC, the typing judgment for a module mod assumes that the names of mod's abstract types have already been forward-declared (*i.e.,* created ahead of time, potentially in an earlier scope), and that they will be passed in as input to the typing judgment. For example, in typing the sealed recursive module above, RMC would assume that in the context there already exists a type variable, say α, which was forward-declared to represent the abstract type component `t`.

Secondly, when typechecking a recursive or sealed module, RMC employs a *two-pass* algorithm. The first pass is a "static" pass, which computes the type components of the module (*e.g.,* discovering that `t` in our example is defined internally as `int`). The information from the static pass is then incorporated into the typing context during the second "main" pass, which fully typechecks the module. In our example, this would mean that the body of the sealed module will be fully typechecked in a context where (1) `X.t` is transparently equal to `int`, and (2) any occurrence of α (the forward-declared representative of `t`) in the typing context is replaced by `int`. This approach successfully avoids double vision by ensuring that all forward references to the abstract type `t` in the typing context are "up-to-date" with the most precise information available about `t` in the current scope.

We adopt the same cure for double vision in MixML, forward-declaring abstract types and performing two passes on the r.h.s. of every linking operation. Unfortunately, since linking involves bidirectional merging instead of unidirectional matching, the RMC solution *per se* is not quite enough.

Cross-Eyed Double Vision Consider the following example:

$$\left(\texttt{X} = \left\{\begin{array}{l} \texttt{t}\triangleright\texttt{t}_1 = \texttt{[:0]},\\ \texttt{u}\triangleright\texttt{u}_1 = \texttt{[int]},\\ \texttt{f} = \texttt{[}\lambda\texttt{x:t}_1\texttt{.x]}\end{array}\right\}\right) \;\texttt{with}\; \left\{\begin{array}{l} \texttt{t}\triangleright\texttt{t}_2 = \texttt{[bool]},\\ \texttt{u}\triangleright\texttt{u}_2 = \texttt{[:0]},\\ \texttt{g} = \texttt{[}\lambda\texttt{y:u}_2\texttt{.X.f(y>1)]}\end{array}\right\}$$

Inside the definition of `g`, both `t` and `u` are accessible under two distinct paths ($\texttt{t}_2$ vs. `X.t` and $\texttt{u}_2$ vs. `X.u`, respectively). Thus, to typecheck the definition, two instances of double vision have to be handled: checking `y>1` requires knowing that $\texttt{u}_2 = \texttt{X.u}$ (and thus $\texttt{u}_2 = \texttt{int}$), and checking the application of `X.f` to the resulting boolean requires knowing that $\texttt{X.t} = \texttt{t}_2$ (and thus $\texttt{X.t} = \texttt{bool}$).

In short, this example suffers from *cross-eyed double vision*. The RMC solution takes care of one direction ($\texttt{X.t} = \texttt{bool}$) but not the other. In order to inform the typechecker that $\texttt{u}_2 = \texttt{int}$, we generalize the RMC approach as follows. In addition to taking as input a list of type variables corresponding to the abstract export types of a module, the MixML module typing judgment takes as input an *import realizer*, which maps the type imports of the module to the concrete types that will instantiate them. In the above example, when performing the main pass on the r.h.s. module, the typechecker will pass in a realizer mapping `u` to `int`, and this information will get propagated to the r.h.s. definition of `u`. In order to compute this realizer, we perform bidirectional type lookup in between the static and main passes of typechecking.

Information about import types is only propagated rightward. For instance, in the above example, the l.h.s. module does not get to know that $\texttt{t}_1 = \texttt{bool}$. This does not incur double vision because the l.h.s. module does not have a name by which to refer to the r.h.s. module. If a module's type imports are not instantiated by (the l.h.s. of) any enclosing linking operation, the typechecker will pass in a realizer that maps them to abstract type variables. For the details of how those import type variables are managed, see Section 4.

While double vision is a problem with no easy workarounds, the seriousness of cross-eyed double vision is somewhat debatable. For instance, in our example, we could easily avoid double vision for the `u` component by making its definition in the r.h.s. module manifestly equal to `X.u`. However, such a workaround is quite brittle. It only works if `u` is an export in the l.h.s. module; otherwise, the definition of $\texttt{u}_2$ as `X.u` is indistinguishable from a transparent type cycle. It is simpler for the programmer to be able to rely on the type system to avoid double vision in both directions.

Opaquely Recursive Types We conclude this section by explaining how to encode ML-style iso-recursive (or "opaquely recursive") `datatype`'s. The encoding is interesting because it demonstrates an instance where we *want* to incur double vision! Luckily, the MixML type system provides a very simple way of manually overriding the built-in solution to double vision.

First, consider the following encodings of specifications and definitions for *non-recursive* `datatype`'s, respectively:

$$\begin{array}{lcl} \{:\ell \approx \mathit{tyc}\} & \overset{\text{def}}{=} & \{\ell \triangleright \mathrm{X} = \texttt{[:}0\texttt{]}, \ell\texttt{\_in} = \texttt{[:}\,\mathit{tyc} \to \mathrm{X}\texttt{]}, \\ & & \qquad \ell\texttt{\_out} = \texttt{[:}\mathrm{X} \to \mathit{tyc}\texttt{]}\} \\ \{\ell \approx \mathit{tyc}\} & \overset{\text{def}}{=} & \{:\ell \approx \mathit{tyc}\}\ \texttt{seals} \\ & & \{\ell \triangleright \mathrm{X} = \texttt{[}\mathit{tyc}\texttt{]}, \ell\texttt{\_in} = \texttt{[}\lambda x{:}\mathrm{X}.x\texttt{]}, \\ & & \qquad \ell\texttt{\_out} = \texttt{[}\lambda x{:}\mathrm{X}.x\texttt{]}\} \end{array}$$

Similar to the interpretations given by Harper and Stone [16] and Dreyer [7], these encodings model the `datatype` definition $\{\ell \approx \mathit{tyc}\}$ as an ADT providing an abstract type ℓ, together with $\ell\texttt{\_in}$ (fold) and $\ell\texttt{\_out}$ (unfold) functions to coerce between ℓ and its underlying representation *tyc*. For brevity, we have only shown the encoding of monomorphic `datatype`'s (of kind 0) here; it easily generalizes to the polymorphic case.

Given these definitions, it would seem straightforward to encode a recursive `datatype` by enclosing a non-recursive `datatype` in a recursive module. For example, integer lists:

$$\texttt{rec}\,(\texttt{X}:\{\texttt{list} = \texttt{[:0]}\})\,\{\texttt{list} \approx \texttt{unit} + \texttt{int} \times \texttt{X.list}\}$$

Unfortunately, this encoding does not typecheck. The reason is that, when the typechecker descends into the body of the sealed `datatype` module, it will (1) discover that `list` is defined to be $\tau = \texttt{unit} + \texttt{int} \times \texttt{X.list}$, (2) try to update the typing context so that `X.list` is transparently equal to τ, and (3) report the presence of a transparent type cycle.

What we want, then, is to be able to switch off the typechecker's double vision avoidance mechanism. We can achieve this by inserting a *unit* suspension/instantiation β-redex:

$$\texttt{rec}\,(\texttt{X}:\{\texttt{list} = \texttt{[:0]}\})\,\texttt{new[}\{\texttt{list} \approx \texttt{unit} + \texttt{int} \times \texttt{X.list}\}\texttt{]}$$

Inserting "$\texttt{new[}\cdot\texttt{]}$" has the effect of dislocating the `datatype` module from its surrounding scope. Units in MixML were designed for encapsulation and separate compilation, and thus the MixML typechecker does not make any attempt to connect the abstract types defined inside a unit (in this case, `list`) with any forward-declared types in the typing context (`X.list`). One could make the case for an alternative design in which the typechecker does attempt to connect the types, but the benefit of being able to switch off double vision avoidance is evident from the above encoding.

$$\begin{array}{llcl} \text{Type Constructors} & \mathrm{A}, \mathrm{B}, \tau & ::= & \alpha \mid \lambda(\overline{\alpha}).\tau \mid \mathrm{A}(\overline{\tau}) \mid \ldots \\ \text{Module Signatures} & \Sigma & ::= & \llbracket = \mathrm{A} \rrbracket \mid \llbracket \tau \rrbracket^{\pm} \mid \llbracket \Phi \rrbracket^{+} \mid \{\!|\overline{\ell : \Sigma}|\!\} \\ \text{Unit Signatures} & \Phi & ::= & \forall \overline{\alpha}.\, \exists \overline{\beta}.\, (\mathcal{L}; \Sigma) \\ \text{Type Substitutions} & \delta & ::= & \{\overline{\alpha \mapsto \mathrm{A}}\} \\ \text{Type Locators} & \mathcal{L} & ::= & \llbracket = \alpha \rrbracket \mid \{\!|\overline{\ell : \mathcal{L}}|\!\} \\ \text{Import Realizers} & \mathcal{R} & ::= & \llbracket = \mathrm{A} \rrbracket \mid \{\!|\overline{\ell : \mathcal{R}}|\!\} \\ \text{Module Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, \mathrm{X} : \Sigma \end{array}$$

$$\begin{array}{rcl} \Sigma.\ell s & \overset{\text{def}}{=} & \begin{cases} \Sigma & \text{if } \ell s = \epsilon \\ \Sigma' & \text{if } \ell s = \ell s'.\ell \text{ and } \Sigma.\ell s' = \{\!|\ell : \Sigma', \ldots|\!\} \\ \uparrow & \text{otherwise} \end{cases} \\ \Sigma(\ell s) & \overset{\text{def}}{=} & \mathrm{A} \ \text{ if } \Sigma.\ell s = \llbracket = \mathrm{A} \rrbracket \\ \mathcal{L}(\alpha) & \overset{\text{def}}{=} & \ell s \ \text{ if } \mathcal{L}(\ell s) = \alpha \text{ and } \nexists\, \ell s' \text{ such that } \mathcal{L}(\ell s') = \alpha \\ \mathcal{R} \subseteq \Sigma & \overset{\text{def}}{=} & \forall\, \ell s, \mathrm{A}.\ \mathcal{R}(\ell s) = \mathrm{A} \Rightarrow \Sigma(\ell s) = \mathrm{A} \\ \mathcal{R}_1 \cup \mathcal{R}_2 & \overset{\text{def}}{=} & \mathcal{R} \text{ such that } \forall \ell s, \mathrm{A}. \\ & & \quad \mathcal{R}(\ell s) = \mathrm{A} \Leftrightarrow (\mathcal{R}_1(\ell s) = \mathrm{A} \vee \mathcal{R}_2(\ell s) = \mathrm{A}) \end{array}$$

Figure 2. Semantic Objects and Auxiliary Definitions

4. The MixML Type System

4.1 Semantic Objects

The MixML type system is based to a large extent on Dreyer's RMC type system for recursive modules [7]. RMC in turn inherits many aspects from the Definition of Standard ML [24]. In particular, it represents the types of modules by *semantic objects*. As in RMC, our semantic objects—shown in Figure 2—are actually signatures from a simpler "internal" type system, enriched with annotations that guide typechecking.[1] These semantic signatures (Σ) include structure signatures ($\{\!|\overline{\ell : \Sigma}|\!\}$), as well as atomic signatures for type modules ($\llbracket = \mathrm{A} \rrbracket$), term modules ($\llbracket \tau \rrbracket^{\pm}$), and units ($\llbracket \Phi \rrbracket^{+}$). Our semantic objects differ from RMC's in that atomic term and unit signatures are annotated with variances, in order to denote whether they are imports ($-$) or exports ($+$).[2] The import/export distinction for type components is handled differently, as we explain below.

A unit signature (Φ) is a module signature that has been universally quantified over the module's import types and existentially quantified over its abstract export types. Unit signatures also contain a *type locator* $\mathcal{L}$ that maps the import names $\overline{\alpha}$ to label paths in Σ. Type locators are used to implement type lookup.

Locators are a syntactic subcategory of *import realizers* $\mathcal{R}$, described in Section 3, which are in turn a subcategory of module signatures Σ. This conveniently allows the sharing of meta-notation for all three kinds of objects, as shown in Figure 2. In particular, all three may be viewed as functions mapping the pathnames of type components to the type components themselves. Well-formed locators have the additional property that all their type components are distinct type variables, and thus they can be viewed as bijective functions between those type variables and their corresponding paths. As a matter of simplicity, we implicitly identify all realizers that represent the same mapping from paths to types, in effect ignoring syntactic differences with respect to empty substructures.

As a concrete example, consider the following unit:

$$\left[\left\{\begin{array}{l} \texttt{A} \triangleright \texttt{A} \ = \ \{\texttt{t} = \texttt{[:0]}\}, \\ \texttt{R} \ = \ \left\{\begin{array}{l} \texttt{u} \triangleright \texttt{u} = \texttt{[:0]}, \\ \texttt{f} = \texttt{[:A.t} \to \texttt{u]} \end{array}\right\} \texttt{seals} \left\{\begin{array}{l} \texttt{u} = \texttt{[int]}, \\ \texttt{f} = \texttt{[}\lambda\texttt{x:(A.t).7]} \end{array}\right\} \end{array}\right\}\right]$$

[1] The internal type system is defined in the expanded online report [11], but is not required in order to understand the static semantics of MixML.

[2] Here we present only the fragment of MixML without unit imports $\llbracket \Phi \rrbracket^{-}$. Higher-order units are discussed in Section 5.

The following semantic signature describes this unit:

$$\forall\alpha.\,\exists\beta.\,(\mathcal{L}; \{\!|\mathtt{A}: \{\!|\mathtt{t}: [\![=\alpha]\!]|\!\}, \mathtt{R}: \{\!|\mathtt{u}: [\![=\beta]\!], \mathtt{f}: [\![\alpha \to \beta]\!]^{+}|\!\}|\!\})$$

where $\mathcal{L}$ is the locator $\{\!|\mathtt{A}: \{\!|\mathtt{t}: [\![=\alpha]\!]|\!\}|\!\}$, mapping α to `A.t`.

Semantic core-level types include standard type functions and application, plus an unspecified set of additional base types. For convenience, we assume that the set of type variables is partitioned into different kinds. This allows us to drop kind annotations from types and type variables, since they can always be derived syntactically. We write $\vdash \mathrm{A} : \mathrm{K}$ to assert that a constructor A has kind K. For type substitutions δ we demand implicitly that they be kind-preserving. We also assume and maintain the invariant that types are kept in β-normal η-long form; we assume substitutions are implicitly normalizing; and we assume the existence of a strict total ordering $<_{Paths}$ on label paths. Following the Definition of SML [24], we employ the convention that a type variable is synonymous (where appropriate) with its η-long form.

4.2 Typing Rules

Figure 3 shows the typing rules for MixML.

The main typing judgment for MixML modules has the form $\Gamma; \mathcal{R}; \overline{\beta} \vdash mod : \Sigma$. Here, $\overline{\beta}$ is a list of distinct type variables representing *mod*'s abstract type *exports*, while the realizer $\mathcal{R}$ captures *mod*'s type *imports*. Note that, due to forward declarations of *mod*'s abstract types arising from linking, the variables $\overline{\beta}$ may also appear free in Γ.

As explained in Section 3, our typing judgment is similar to the one in RMC, but more general due to the inclusion of realizers. Another minor difference from RMC is that we choose (for conciseness) to write $\overline{\beta}$ to the left of the turnstile instead of writing "with $\overline{\beta}\downarrow$" at the right end of the judgment. Just as in RMC, the MixML type system tracks type definitions linearly, ensuring that each β in $\overline{\beta}$ gets defined by *mod* exactly once. Thus, although $\overline{\beta}$ in the present judgment is treated like a linear list of capabilities (as opposed to the formulation in RMC, in which "with $\overline{\beta}\downarrow$" was treated as a *type effect*), the underlying semantics is morally the same, and we consider the difference to be essentially cosmetic.

Thanks to the implicit kinding of type variables, type contexts degenerate into simple sets. Because a suitable type context Δ binding all free type variables in a judgment is trivially inferable, we omit the Δ's in the presentation of our rules. We write $\overline{\alpha}$ fresh in the premise of a rule to mean that none of the $\overline{\alpha}$ appears in the context of the conclusion of the rule.

Following RMC, we use shading of certain premises in the typing rules to denote the delta between the main typing judgment and the *static* typing judgment ($\vdash_{\overline{\mathrm{stat}}}$). The static judgment is used to implement the static pass of recursive linking, as described in Section 3. To obtain the static version of any rule, simply remove all shaded premises, replace all $\vdash$'s with $\vdash_{\overline{\mathrm{stat}}}$'s, and erase all occurrences of term signatures $[\![\tau]\!]^{\pm}$ to the empty signature $\{\!||\!\}$.

Most of the rules for basic modules are fairly straightforward. Notably, Rule 1 for variables X returns the signature $|\Sigma|$, which turns all imports of Σ into exports. The reason is that, regardless of whether the module that X is bound to—call it *mod*—is fully defined, the module expression X is a *definite* reference to *mod* and is therefore itself fully defined. To take a concrete example, recall the encoding of sig_1 `matches` sig_2 in Section 2. In the encoding, we bind sig_1 to a variable X, and then check that the linking of X with sig_2 results in a complete module. This only works because X will *export* a component corresponding to each of sig_1's imports, and those exports will be used to satisfy all the imports of sig_2.

Rule 3 for `[:K]` uses the import realizer to look up the definition of its type import (which is A). The other rules for type and term modules are straightforward. Note that Rules 5 and 6 have shaded premises because terms are ignored during the static pass.

Typing namespaces (Rule 7) and projection (Rule 8) is also straightforward. The latter rule requires the signatures of all components other than the one being projected out to be of the form $|\Sigma|$. This enforces that no term imports (of signature $[\![\tau]\!]^{-}$) are being hidden. (The reasons for enforcing this are discussed in Section 2.) The hidden components cannot contain type imports either, as the import realizer in the premise contains only the projected label ℓ.

Rule 9 handles recursive linking. It is the central rule of our system, and while it is admittedly somewhat complex, it is essentially just a bidirectional generalization of RMC's typing rule for recursive modules. Let us first step through the rule ignoring the $\mathcal{L}$'s and $\mathcal{R}$'s. Type checking proceeds by first checking mod_1, producing a signature Σ_1 for X. As in RMC, checking mod_2 requires two passes. The first, "static" pass, only collects *type* specifications and definitions from mod_2. The linking rule then uses bidirectional lookup (Rule 18) to look up mod_1's type imports in mod_2 and vice versa. (RMC only employs unidirectional lookup.)

The bidirectional lookup judgment will fail if it detects any transparent type cycles. Assuming it succeeds, it yields a type substitution δ, which is then applied to the signature Σ_1 previously computed for mod_1, intuitively "patching" it with the appropriate type definitions from mod_2. In this way, when we typecheck mod_2 fully in the subsequent "main" pass, we see no difference between mod_2's type components and the components with the same name in X. This is the key to avoiding double vision. Lastly, the signatures of mod_1 and mod_2 are merged, yielding the final signature Σ. Merging is defined by a straightforward auxiliary judgment.

Now about those $\mathcal{L}$'s and $\mathcal{R}$'s: To deal with type imports and cross-eyed double vision, the linking rule has to properly adjust locators and realizers as it proceeds. The input realizer is first split into $\mathcal{R}_1$ and $\mathcal{R}_2$, where the former contains imports (of the linked module) stemming from mod_1, and the latter those from mod_2. Note that these may overlap. In addition, each module may have additional *local* imports, *i.e.,* imports that the other module will satisfy. These are handled by locally extending the realizer with fresh locators $\mathcal{L}_1$ and $\mathcal{L}_2$, which are later used for type lookup.

Let us walk through the cross-eyed double vision example from Section 3. Because the linked module has neither imports nor abstract type exports, Rule 9 will be invoked with $\mathcal{R}_1 = \mathcal{R}_2 = \{\!||\!\}$ and $\overline{\beta}_1 = \overline{\beta}_2 = \emptyset$. However, it will introduce fresh variables α_1 and α_2 for the local imports of both sides and define $\mathcal{L}_1 = \{\!|\mathtt{t}: [\![=\alpha_1]\!]|\!\}$ and $\mathcal{L}_2 = \{\!|\mathtt{u}: [\![=\alpha_2]\!]|\!\}$ as local realizers for the l.h.s. and r.h.s., respectively. Traversing into the l.h.s. delivers $\Sigma_1 = \{\!|\mathtt{t}: [\![=\alpha_1]\!], \mathtt{u}: [\![=\mathtt{int}]\!], \mathtt{f}: [\![\alpha_1 \to \alpha_1]\!]^{+}|\!\}$. Next, the static pass on the r.h.s. is performed, yielding $\Sigma_2 = \{\!|\mathtt{t}: [\![=\mathtt{bool}]\!], \mathtt{u}: [\![=\alpha_2]\!], \mathtt{g}: \{\!||\!\}|\!\}$. Note that `g` has empty signature in Σ_2 since the static pass ignores term modules.

Using Σ_1, Σ_2, $\mathcal{L}_1$, and $\mathcal{L}_2$, bidirectional type lookup returns the substitution $\delta = \{\alpha_1 \mapsto \mathtt{bool}, \alpha_2 \mapsto \mathtt{int}\}$. For the main pass on the r.h.s., δ is applied to Σ_1 and the locator $\mathcal{L}_2$, turning the latter into the realizer $\{\!|\mathtt{u}: [\![=\mathtt{int}]\!]|\!\}$. Thus, in this pass, the typechecker will see that `X.t` is `bool`, and it will know to implement `u` as `int`, thus avoiding cross-eyed double vision. Finally, it will return the signature $\Sigma_2' = \{\!|\mathtt{t}: [\![=\mathtt{bool}]\!], \mathtt{u}: [\![=\mathtt{int}]\!], \mathtt{g}: [\![\mathtt{int} \to \mathtt{bool}]\!]^{+}|\!\}$, which can be merged successfully with $\delta\Sigma_1$.

Rule 10 for opaque linking is very similar to Rule 9, but slightly simpler because the result of opaque linking is not permitted to have any residual imports. (This is enforced by replacing $\mathcal{R}_1$ and $\mathcal{R}_2$ with $\{\!||\!\}$, and requiring that the merged signature have the form $|\Sigma|$.) In addition, the type imports of mod_1 (the $\overline{\alpha_1}$), which mod_2 must satisfy, become abstract type *exports* for the whole module, while the abstract type exports of mod_2 (the $\overline{\beta_2}$), are fresh variables only introduced into scope locally (since mod_2 is hidden). Finally, note that the final signature of the module is derived solely from the signature of mod_1—all information about mod_2 is kept secret.

$$\begin{array}{lcllcl} |\llbracket = \mathrm{A}\rrbracket| & \stackrel{\text{def}}{=} & \llbracket = \mathrm{A}\rrbracket & |\llbracket\Phi\rrbracket^+| & \stackrel{\text{def}}{=} & \llbracket\Phi\rrbracket^+ \\ |\llbracket\tau\rrbracket^\pm| & \stackrel{\text{def}}{=} & \llbracket\tau\rrbracket^+ & |\{\!\!|\overline{\ell:\Sigma}|\!\!\}| & \stackrel{\text{def}}{=} & \{\!\!|\overline{\ell:|\Sigma|}|\!\!\} \end{array}$$

$\boxed{\textbf{Modules: } \Gamma;\mathcal{R};\overline{\beta}\vdash \mathit{mod} : \Sigma}$

$$\frac{\mathrm{X}:\Sigma\in\Gamma}{\Gamma;\{\!\!||\!\!\};\emptyset\vdash \mathrm{X} : |\Sigma|}\;(1) \qquad \frac{}{\Gamma;\{\!\!||\!\!\};\emptyset\vdash \{\} : \{\!\!||\!\!\}}\;(2) \qquad \frac{\vdash \mathrm{A}:\mathrm{K}}{\Gamma;\llbracket=\mathrm{A}\rrbracket;\emptyset\vdash \texttt{[:K]} : \llbracket=\mathrm{A}\rrbracket}\;(3) \qquad \frac{\Gamma\vdash \mathit{tyc}\leadsto \mathrm{A}:\mathrm{K}}{\Gamma;\{\!\!||\!\!\};\emptyset\vdash \texttt{[}\mathit{tyc}\texttt{]} : \llbracket=\mathrm{A}\rrbracket}\;(4)$$

$$\frac{\Gamma\vdash \mathit{tyc}\leadsto \tau : 0}{\Gamma;\{\!\!||\!\!\};\emptyset\vdash \texttt{[:}\mathit{tyc}\texttt{]} : \llbracket\tau\rrbracket^-}\;(5) \qquad \frac{\Gamma\vdash \mathit{exp} : \tau}{\Gamma;\{\!\!||\!\!\};\emptyset\vdash \texttt{[}\mathit{exp}\texttt{]} : \llbracket\tau\rrbracket^+}\;(6)$$

$$\frac{\Gamma;\mathcal{R};\overline{\beta}\vdash \mathit{mod} : \Sigma}{\Gamma;\{\!\!|\ell:\mathcal{R}|\!\!\};\overline{\beta}\vdash \{\ell=\mathit{mod}\} : \{\!\!|\ell:\Sigma|\!\!\}}\;(7) \qquad \frac{\Gamma;\{\!\!|\ell:\mathcal{R}|\!\!\};\overline{\beta}\vdash \mathit{mod} : \{\!\!|\ell:\Sigma,\overline{\ell':|\Sigma'|}|\!\!\}}{\Gamma;\mathcal{R};\overline{\beta}\vdash \mathit{mod}.\ell : \Sigma}\;(8)$$

$$\frac{\begin{array}{c}\vdash\mathcal{L}_1\ \mathsf{locates}\ \overline{\alpha_1} \qquad \Gamma;\mathcal{L}_1\uplus\mathcal{R}_1;\overline{\beta_1}\vdash \mathit{mod}_1 : \Sigma_1 \\ \vdash\mathcal{L}_2\ \mathsf{locates}\ \overline{\alpha_2} \qquad \Gamma,\mathrm{X}:\Sigma_1;\mathcal{L}_2\uplus\mathcal{R}_2;\overline{\beta_2}\vdash_{\mathsf{stat}} \mathit{mod}_2 : \Sigma_2 \qquad \overline{\alpha_1},\overline{\alpha_2}\ \mathsf{fresh} \\ \vdash(\mathcal{L}_1;\Sigma_1)\rightleftarrows(\mathcal{L}_2;\Sigma_2)\leadsto\delta \qquad \Gamma,\mathrm{X}:\delta\Sigma_1;\delta\mathcal{L}_2\uplus\mathcal{R}_2;\overline{\beta_2}\vdash \mathit{mod}_2 : \Sigma_2' \qquad \vdash\delta\Sigma_1+\Sigma_2'\Rightarrow\Sigma\end{array}}{\Gamma;\mathcal{R}_1\cup\mathcal{R}_2;\overline{\beta_1},\overline{\beta_2}\vdash (\mathrm{X}=\mathit{mod}_1)\ \texttt{with}\ \mathit{mod}_2 : \Sigma}\;(9)$$

$$\frac{\begin{array}{c}\vdash\mathcal{L}_1\ \mathsf{locates}\ \overline{\alpha_1} \qquad \Gamma;\mathcal{L}_1;\overline{\beta_1}\vdash \mathit{mod}_1 : \Sigma_1 \\ \vdash\mathcal{L}_2\ \mathsf{locates}\ \overline{\alpha_2} \qquad \Gamma,\mathrm{X}:\Sigma_1;\mathcal{L}_2;\overline{\beta_2}\vdash_{\mathsf{stat}} \mathit{mod}_2 : \Sigma_2 \qquad \overline{\beta_2},\overline{\alpha_2}\ \mathsf{fresh} \\ \vdash(\mathcal{L}_1;\Sigma_1)\rightleftarrows(\mathcal{L}_2;\Sigma_2)\leadsto\delta \qquad \delta\Gamma,\mathrm{X}:\delta\Sigma_1;\delta\mathcal{L}_2;\overline{\beta_2}\vdash \mathit{mod}_2 : \Sigma_2' \qquad \vdash\delta\Sigma_1+\Sigma_2'\Rightarrow|\Sigma|\end{array}}{\Gamma;\{\!\!||\!\!\};\overline{\beta_1},\overline{\alpha_1}\vdash (\mathrm{X}=\mathit{mod}_1)\ \texttt{seals}\ \mathit{mod}_2 : |\Sigma_1|}\;(10)$$

$$\frac{\Gamma\vdash \mathit{mod} : \Phi}{\Gamma;\{\!\!||\!\!\};\emptyset\vdash \texttt{[}\mathit{mod}\texttt{]} : \llbracket\Phi\rrbracket^+}\;(11) \qquad \frac{\Gamma\vdash \mathit{mod} : \llbracket\forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\Sigma)\rrbracket^+ \qquad \mathsf{dom}(\delta)=\{\overline{\alpha},\overline{\beta}\}}{\Gamma;\delta\mathcal{L};\delta\overline{\beta}\vdash \texttt{new}\ \mathit{mod} : \delta\Sigma}\;(12)$$

$\boxed{\textbf{Complete Modules and Units: } \Gamma\vdash \mathit{mod} : \Sigma \quad \Gamma\vdash \mathit{mod} : \Phi}$

$$\frac{\Gamma;\{\!\!||\!\!\};\overline{\beta}\vdash \mathit{mod} : |\Sigma| \qquad \overline{\beta}\ \mathsf{fresh} \qquad \overline{\beta}\notin\mathsf{FV}(\Sigma)}{\Gamma\vdash \mathit{mod} : |\Sigma|}\;(13) \qquad \frac{\Gamma;\mathcal{L};\overline{\beta}\vdash \mathit{mod} : \Sigma \qquad \vdash\mathcal{L}\ \mathsf{locates}\ \overline{\alpha} \qquad \overline{\alpha},\overline{\beta}\ \mathsf{fresh}}{\Gamma\vdash \mathit{mod} : \forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\Sigma)}\;(14)$$

$\boxed{\textbf{Core-Language Types and Terms: } \Gamma\vdash \mathit{tyc}\leadsto\mathrm{A}:\mathrm{K} \quad \Gamma\vdash \mathit{exp} : \tau}$

$$\frac{\Gamma\vdash \mathit{mod} : \llbracket=\mathrm{A}\rrbracket \qquad \vdash\mathrm{A}:\mathrm{K}}{\Gamma\vdash \texttt{Tyc}(\mathit{mod})\leadsto\mathrm{A}:\mathrm{K}}\;(15) \qquad \frac{\Gamma\vdash \mathit{mod} : \llbracket\tau\rrbracket^+}{\Gamma\vdash \texttt{Val}(\mathit{mod}) : \tau}\;(16)$$

Rules for α, $\lambda(\overline{\alpha}).\mathit{tyc}$, $\mathit{tyc}(\overline{\mathit{tyc}})$ are standard.

$\boxed{\textbf{Type Locators: } \vdash\mathcal{L}\ \mathsf{locates}\ \overline{\alpha}}$

$$\frac{\overline{\alpha}=\alpha_1,\ldots,\alpha_n \qquad \mathsf{range}(\mathcal{L})=\{\overline{\alpha}\} \qquad \mathcal{L}\ \text{is bijective} \qquad \forall i,j\ (\text{s.t. } i<j)\in 1..n:\ \mathcal{L}(\alpha_i)<_{\mathit{Paths}}\mathcal{L}(\alpha_j)}{\vdash\mathcal{L}\ \mathsf{locates}\ \overline{\alpha}}\;(17)$$

$\boxed{\textbf{Bidirectional Type Lookup: } \vdash(\mathcal{L}_1;\Sigma_1)\rightleftarrows(\mathcal{L}_2;\Sigma_2)\leadsto\delta}$

$$\frac{\begin{array}{c}(\Sigma_2\circ\mathcal{L}_1)\uplus(\Sigma_1\circ\mathcal{L}_2)=\{\alpha_1\mapsto\mathrm{A}_1,\ldots,\alpha_n\mapsto\mathrm{A}_n\} \\ \delta_0=\{\} \qquad \forall i,j\ (\text{s.t. } i\le j)\in 1..n:\quad \alpha_j\notin\mathsf{FV}(\mathrm{A}_i) \quad \delta_i=\delta_{i-1}\uplus\{\alpha_i\mapsto\delta_{i-1}\mathrm{A}_i\}\end{array}}{\vdash(\mathcal{L}_1;\Sigma_1)\rightleftarrows(\mathcal{L}_2;\Sigma_2)\leadsto\delta_n}\;(18)$$

$\boxed{\textbf{Signature Merging: } \vdash\Sigma_1+\Sigma_2\Rightarrow\Sigma}$

$$\frac{\vdash\Sigma_2+\Sigma_1\Rightarrow\Sigma}{\vdash\Sigma_1+\Sigma_2\Rightarrow\Sigma}\;(19) \qquad \frac{}{\vdash\llbracket=\mathrm{A}\rrbracket+\llbracket=\mathrm{A}\rrbracket\Rightarrow\llbracket=\mathrm{A}\rrbracket}\;(20) \qquad \frac{\vdash\tau_1\le\tau_2}{\vdash\llbracket\tau_1\rrbracket^\pm+\llbracket\tau_2\rrbracket^-\Rightarrow\llbracket\tau_1\rrbracket^\pm}\;(21) \qquad \frac{}{\vdash\Sigma+\{\!\!||\!\!\}\Rightarrow\Sigma}\;(22)$$

$$\frac{\ell\notin\overline{\ell_2} \qquad \vdash\{\!\!|\overline{\ell_1:\Sigma_1}|\!\!\}+\{\!\!|\overline{\ell_2:\Sigma_2}|\!\!\}\Rightarrow\{\!\!|\overline{\ell_3:\Sigma_3}|\!\!\}}{\vdash\{\!\!|\ell:\Sigma,\overline{\ell_1:\Sigma_1}|\!\!\}+\{\!\!|\overline{\ell_2:\Sigma_2}|\!\!\}\Rightarrow\{\!\!|\ell:\Sigma,\overline{\ell_3:\Sigma_3}|\!\!\}}\;(23) \qquad \frac{\vdash\Sigma_1+\Sigma_2\Rightarrow\Sigma_3 \qquad \vdash\{\!\!|\overline{\ell_1:\Sigma_1'}|\!\!\}+\{\!\!|\overline{\ell_2:\Sigma_2'}|\!\!\}\Rightarrow\{\!\!|\overline{\ell_3:\Sigma_3'}|\!\!\}}{\vdash\{\!\!|\ell:\Sigma_1,\overline{\ell_1:\Sigma_1'}|\!\!\}+\{\!\!|\ell:\Sigma_2,\overline{\ell_2:\Sigma_2'}|\!\!\}\Rightarrow\{\!\!|\ell:\Sigma_3,\overline{\ell_3:\Sigma_3'}|\!\!\}}\;(24)$$

Figure 3. Typing Rules for MixML

Consider the following example of opaque linking:

$$\left\{\begin{array}{lcl}\mathtt{t}\triangleright\mathtt{t}_1 & = & \mathtt{[:0]},\\ \mathtt{u}\triangleright\mathtt{u}_1 & = & \mathtt{[int]},\\ \mathtt{f} & = & \mathtt{[:u_1}\rightarrow\mathtt{t_1]}\end{array}\right\}\ \mathtt{seals}\ \left\{\begin{array}{lcl}\mathtt{t}\triangleright\mathtt{t}_2 & = & \mathtt{[bool]},\\ \mathtt{u}\triangleright\mathtt{u}_2 & = & \mathtt{[:0]},\\ \mathtt{f} & = & \mathtt{[}\lambda\mathtt{x:u_2.true]}\end{array}\right\}$$

The linked module creates a single abstract export type, say α. Neither constituent module creates any abstract types independent of the sealing, so Rule 10 is applied with $\overline{\beta_1} = \overline{\beta_2} = \emptyset$ and $\overline{\alpha_1} = \alpha$. Then, $\Sigma_1 = \{\!|\mathtt{t}:[\![=\alpha]\!], \mathtt{u}:[\![=\mathtt{int}]\!], \mathtt{f}:[\![\mathtt{int}\rightarrow\alpha]\!]^-|\!\}$ is derived for the l.h.s. The r.h.s. locally imports $\mathtt{u}$, so we choose a single fresh α_2 and $\Sigma_2 = \{\!|\mathtt{t}:[\![=\mathtt{bool}]\!], \mathtt{u}:[\![=\alpha_2]\!], \mathtt{f}:\{\!||\!\}|\!\}$ will be returned by the static pass. With lookup returning $\delta = \{\alpha\mapsto\mathtt{bool}, \alpha_2\mapsto\mathtt{int}\}$, the main pass proceeds under context $\delta\Gamma$, where any forward references to α are replaced by $\mathtt{bool}$. The main pass yields $\Sigma'_2 = \{\!|\mathtt{t}:[\![=\mathtt{bool}]\!], \mathtt{u}:[\![=\mathtt{int}]\!], \mathtt{f}:[\![\mathtt{int}\rightarrow\mathtt{bool}]\!]^+|\!\}$. Merging $\delta\Sigma_1$ and Σ'_2 produces $\Sigma = \Sigma'_2$, and since Σ has no residual imports, we have $\Sigma = |\Sigma|$. In contrast to Rule 9, Σ is not taken as the final signature. Instead, the signature $|\Sigma_1| = \{\!|\mathtt{t}:[\![=\alpha]\!], \mathtt{u}:[\![=\mathtt{int}]\!], \mathtt{f}:[\![\mathtt{int}\rightarrow\alpha]\!]^+|\!\}$ is returned. Note how it keeps $\mathtt{t}$ abstract (using the abstract type name α passed in from the context), while marking $\mathtt{f}$ as an export.

Rules 11 and 12, dealing with unit introduction and elimination, are very simple. The former invokes the unit typing judgment described below. The latter instantiates the given unit by choosing an appropriate substitution δ for the unit's import and export type names, and then applying δ to the signature Σ of the unit's constituent module. Although the rule appears nondeterministic, the choice of δ is in fact completely determined by the import realizer and abstract type export list, which are inputs to typechecking.

Projection of (core) types and terms (Rules 15 and 16) requires the module being projected from to be *complete* (no imports). This prevents meaningless examples like $\mathtt{Tyc([:0])}$ or $\mathtt{Val([:int])}$. Note that projections of the form $\mathtt{Tyc}(\mathrm{X}.\ell s)$ or $\mathtt{Val}(\mathrm{X}.\ell s)$ are always acceptable, because variables are definite references (see above). The latter may, however, raise a runtime "blackhole" exception if the component $\mathrm{X}.\ell s$ refers to is as yet undefined.

Module completeness is ensured by Rule 13, which checks that mod neither has type imports (by passing in an empty realizer) nor term imports (the signature is of the form $|\Sigma|$). Local abstract types $\overline{\beta}$ may not escape their scope by appearing in Σ.

Finally, Rule 14 typechecks a module as a self-contained unit. It introduces fresh names for import types ($\overline{\alpha}$) and export types ($\overline{\beta}$), which become quantified in the resulting unit signature. While the rule appears to have to guess the structure of the type locator $\mathcal{L}$, as well as the number, order, and kinds of $\overline{\alpha}$ and $\overline{\beta}$, out of nowhere, there is in fact only one way to choose them, which is easy to compute algorithmically. (See the expanded online report [11] for details.)

4.3 Dynamic Semantics and Type Soundness

The dynamic semantics of MixML is given by evidence translation into a simpler internal language (IL) closely based on Dreyer's RTG calculus for recursive type generativity [9]. As in RMC, soundness of the translation together with soundness of the IL is then sufficient to establish type soundness for MixML.

The evidence translation follows closely the style of Flatt and Felleisen's *unit* language [14, 30]. In particular, the translation employs a backpatching semantics, using lazy reference cells to enable recursive linking for dynamic module components. Thus, a module mod of signature Σ translates to a module-level *function*, whose argument has the same shape as Σ but with uninitialized cells corresponding to the exports of Σ (the imports may or may not be initialized). When called, this function will patch in definitions for those exports. In so doing, the function may attempt to dereference any component of the argument (import or export), which may in turn result in a run-time error if that component has not yet been backpatched. For full details, we refer the reader to the expanded online report [11].

$$\begin{array}{lrcl}\text{Modules} & mod & ::= & \ldots \mid \mathtt{[:}\,usig\mathtt{]}\\ \text{Unit Signatures} & usig & ::= & mod\ \mathtt{import}\ \overline{\ell s} \mid mod\ \mathtt{export}\ \overline{\ell s}\end{array}$$

Figure 4. Higher-Order MixML Syntax Extensions

$$\begin{array}{lrcl}\text{Module Signatures} & \Sigma & ::= & \ldots \mid [\![\Phi]\!]^-\\ \text{Unit Signatures} & \Phi & ::= & \forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L}_1;\mathcal{L}_2;\Sigma)\end{array}$$

$$\begin{array}{rcl}\forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\Sigma) & \overset{\text{def}}{=} & \forall\overline{\alpha}.\,\exists\overline{\beta}.\,(\mathcal{L};\mathcal{L}';\Sigma)\ \text{ for some } \mathcal{L}'\\ \mathsf{prefix?}(\overline{\ell s},\ell s) & \overset{\text{def}}{=} & \exists\ell s_1\in\overline{\ell s}.\,\exists\ell s_2.\ \text{s.t. } \ell s = \ell s_1.\ell s_2\end{array}$$

Figure 5. Higher-Order MixML Semantic Object Extensions

5. Higher-Order MixML

The language presented in the previous section provides only first-order units. In this section we extend it with higher-order units, thus subsuming higher-order modules.

5.1 Syntax

Figure 4 shows the syntactic extensions necessary for supporting higher-order units. Basically, all that is needed is to add atomic unit imports $\mathtt{[:}\,usig\mathtt{]}$. However, to describe a unit import, it is necessary to introduce a new syntactic class $usig$ of *unit signatures*.

A unit signature takes one of two symmetric forms, written $mod\ \mathtt{import}\ \overline{\ell s}$ and $mod\ \mathtt{export}\ \overline{\ell s}$. In both forms, mod is a MixML module representing an ML signature, *i.e.,* it must have neither term exports nor abstract type exports. The `import` and `export` clauses serve to identify which components specified in mod are to be treated as imports and which as exports in the unit that the $usig$ is describing. In the case of $mod\ \mathtt{import}\ \overline{\ell s}$, the list $\overline{\ell s}$ of paths enumerates all components of mod that are to be considered imports, treating all others as exports. Conversely, $mod\ \mathtt{export}\ \overline{\ell s}$ lists the exports, and treats all other components as imports. A path $\ell s \in \overline{\ell s}$ may point to an entire structure in mod, in which case the annotation applies to all its subcomponents.

For example, recall the unit $\mathtt{U_A}$ from the end of Section 2.

$$\mathtt{[}(\mathrm{X}=\mathtt{new}\ \mathrm{S})\ \mathtt{with}\ \{\mathtt{A}=mod_{\mathtt{A}}\}\mathtt{]}$$

We can assign it the following unit signature:

$$(\mathtt{new}\ \mathrm{S})\ \mathtt{export}\ \mathtt{A}$$

Or alternatively:

$$(\mathtt{new}\ \mathrm{S})\ \mathtt{import}\ \mathtt{B}$$

We provide both forms merely as a convenience.

The encoding of the functor F given in Section 2,

$$\lambda(\mathrm{X}{:}sig).mod \ \overset{\text{def}}{=}\ \mathtt{[}\{\mathtt{Arg}\triangleright\mathrm{X}=sig,\ \mathtt{Res}=mod\}\mathtt{]}$$

can be classified with a unit signature as follows (assuming that sig' is a suitable specification of its body mod):

$$\{\mathtt{Arg}\triangleright\mathrm{X}=sig,\ \mathtt{Res}=sig'\}\ \mathtt{import}\ \mathtt{Arg}$$

This corresponds to the ML functor signature $(\mathrm{X}:sig)\rightarrow sig'$.

Unit signature bindings (*i.e.,* bindings of unit signatures $usig$ to signature variables S for convenience) are easy to encode as well. Just as with regular signature bindings, we simply suspend the

$$
\begin{array}{rclrcl}
|[\![=\mathrm{A}]\!]| & \overset{\mathrm{def}}{=} & [\![=\mathrm{A}]\!] & |[\![\Phi]\!]^{\pm}| & \overset{\mathrm{def}}{=} & [\![\Phi]\!]^{+} \\
|[\![\tau]\!]^{\pm}| & \overset{\mathrm{def}}{=} & [\![\tau]\!]^{+} & |\{\!|\overline{\ell:\Sigma}|\!\}| & \overset{\mathrm{def}}{=} & \{\!|\overline{\ell:|\Sigma|}|\!\}
\end{array}
\qquad
\begin{array}{rclrcl}
-[\![=\mathrm{A}]\!] & \overset{\mathrm{def}}{=} & [\![=\mathrm{A}]\!] & -[\![\Phi]\!]^{\pm} & \overset{\mathrm{def}}{=} & [\![\Phi]\!]^{\mp} \\
-[\![\tau]\!]^{\pm} & \overset{\mathrm{def}}{=} & [\![\tau]\!]^{\mp} & -\{\!|\overline{\ell:\Sigma}|\!\} & \overset{\mathrm{def}}{=} & \{\!|\overline{\ell:-\Sigma}|\!\}
\end{array}
$$

$\boxed{\textbf{Modules: } \Gamma;\mathcal{L};\overline{\beta}\vdash \mathit{mod} : \Sigma}$

$$\frac{\Gamma\vdash \mathit{usig} \rightsquigarrow \Phi}{\Gamma;\{\!||\!\};\emptyset\vdash \texttt{[:}\mathit{usig}\texttt{]} : [\![\Phi]\!]^{-}}\ (25)$$

$\boxed{\textbf{Unit Signatures: } \Gamma\vdash \mathit{usig} \rightsquigarrow \Phi}$

$$\frac{\begin{array}{c}\Gamma\vdash \mathit{mod} : \forall\overline{\alpha}.\,\exists\emptyset.\,(\mathcal{L};\Sigma') \qquad |\Sigma| = |\Sigma'| = -\Sigma' \qquad \mathcal{L} = \mathcal{L}_1 \uplus \mathcal{L}_2 \\ \vdash \mathcal{L}_1 \text{ locates } \overline{\alpha_1} \qquad \vdash \mathcal{L}_2 \text{ locates } \overline{\alpha_2} \qquad \{\overline{\alpha_1}\} = \{\alpha \mid \mathsf{prefix?}(\overline{\ell s}, \mathcal{L}(\alpha))\} \\ \forall \ell s\text{: } \Sigma'.\ell s = [\![\tau]\!]^{-} \text{ or } [\![\Phi]\!]^{-} \Rightarrow (\Sigma.\ell s = \Sigma'.\ell s \Leftrightarrow \mathsf{prefix?}(\overline{\ell s}, \ell s)) \qquad \forall \ell s \in \overline{\ell s}\text{: } \Sigma.\ell s \text{ is defined}\end{array}}{\Gamma\vdash \mathit{mod}\ \texttt{import}\ \overline{\ell s} \rightsquigarrow \forall\overline{\alpha_1}.\,\exists\overline{\alpha_2}.\,(\mathcal{L}_1;\mathcal{L}_2;\Sigma)}\ (26)$$

$$\frac{\Gamma\vdash \mathit{mod}\ \texttt{import}\ \overline{\ell s} \rightsquigarrow \forall\overline{\alpha_1}.\,\exists\overline{\alpha_2}.\,(\mathcal{L}_1;\mathcal{L}_2;\Sigma)}{\Gamma\vdash \mathit{mod}\ \texttt{export}\ \overline{\ell s} \rightsquigarrow \forall\overline{\alpha_2}.\,\exists\overline{\alpha_1}.\,(\mathcal{L}_2;\mathcal{L}_1;-\Sigma)}\ (27)$$

$\boxed{\textbf{Signature Merging: } \vdash \Sigma_1 + \Sigma_2 \Rightarrow \Sigma}$

$$\frac{}{\vdash [\![\Phi]\!]^{-} + [\![\Phi]\!]^{-} \Rightarrow [\![\Phi]\!]^{-}}\ (28) \qquad \frac{\vdash \Phi_1 \leq \Phi_2}{\vdash [\![\Phi_1]\!]^{+} + [\![\Phi_2]\!]^{-} \Rightarrow [\![\Phi_1]\!]^{+}}\ (29)$$

$\boxed{\textbf{Unit Signature Matching: } \vdash \Phi_1 \leq \Phi_2}$

$$\frac{\vdash (\mathcal{L}_{11};\Sigma_1) \rightleftarrows (\mathcal{L}_{22};\Sigma_2) \rightsquigarrow \delta \qquad \vdash \delta\Sigma_1 + -\delta\Sigma_2 \Rightarrow |\Sigma|}{\vdash \forall\overline{\alpha_1}.\,\exists\overline{\beta_1}.\,(\mathcal{L}_{11};\mathcal{L}_{12};\Sigma_1) \leq \forall\overline{\alpha_2}.\,\exists\overline{\beta_2}.\,(\mathcal{L}_{21};\mathcal{L}_{22};\Sigma_2)}\ (30)$$

Figure 6. Higher-Order MixML Type System Extensions

unit signature using a unit. That is, we bind S = `[[:`*usig*`]]`. Then, whenever we wish later in the program to create a unit import U of signature *usig*, we simply bind U = `new` S. As units subsume functors, this demonstrates how MixML encodes functor signature bindings, of the kind that exist in higher-order dialects of the ML module system.

5.2 Semantic Objects

In order to be able to express unit imports, the definition of semantic objects has to be extended in two respects, shown in Figure 5:

1. Atomic unit signatures can be marked as imports with a negative polarity as in $[\![\Phi]\!]^{-}$, analogously to atomic term signatures.
2. Unit signatures Φ contain two type locators $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively mapping the import types $\overline{\alpha}$ and abstract export types $\overline{\beta}$. The export locator $\mathcal{L}_2$ is used for higher-order unit signature matching and thus only is needed when representing the translation of MixML-level *usig*'s; we omit it in other places.

5.3 Typing Rules

Figure 6 shows the additional rules (and changes to existing rules) that are necessary to incorporate unit imports.

Rule 25 is the obvious rule for unit imports. Note that Rule 12 for `new` is left unchanged: it still requires that the argument *mod* is a unit export module, *i.e.,* that it actually contains a unit definition. Again, the unit it *contains* is free to have imports, which will become imports of `new` *mod* itself. More concretely, the premise prevents examples like `new [:`*usig*`]` from type checking, which would instantiate a non-existent unit, but permits `new [[:`*usig*`]]` (as seen in the encoding of unit signature bindings given earlier).

The two rules for elaborating unit signatures are simpler than they might look. Rule 26 infers a unit signature for *mod* and requires that it be free of exports (no export type variables, and $|\Sigma'| = -\Sigma'$). It then partitions the components according to the paths listed in $\overline{\ell s}$, turning those reachable from $\overline{\ell s}$ into imports and those unreachable from $\overline{\ell s}$ into exports. The second form of unit signature (with an `export` clause) is handled in the dual manner.

One interesting restriction is that the merging of two unit imports (Rule 28) does not allow their signatures to differ. This restriction is in place in order to ensure decidability. (Without it, the guessing of locators described at the end of Section 4.2 becomes difficult.) The restriction does not place any limitations, though, on the MixML encoding of ML-style higher-order functors, since that encoding will never attempt to merge two unit imports. In contrast, when matching unit exports against unit imports, merging allows subtyping in the form of unit signature matching (Rule 29).

Rule 30 defines unit signature matching in terms of linking. The rule checks whether the exports of Φ_1 subsume the exports of Φ_2 and, contravariantly, the imports of Φ_2 subsume those of Φ_1. It does so by *inverting* the module signature Σ_2 from Φ_2 (*i.e.,* swapping imports and exports) and trying to link it against Σ_1. If this succeeds without any remaining imports, we know that the exports of subsignature Φ_1 in fact match the exports of Φ_2, and conversely for the (contravariant) imports of the two signatures. Type components are dealt with by using the bidirectional lookup judgment to simultaneously look up the type *exports* of Σ_2 in Σ_1 and the *imports* of Σ_1 in Σ_2. Notably, this is the only rule that makes use of export locators. In the case of Φ_2, an export locator is guaranteed to exist because Φ_2 is a target signature—invariants of the type system (formalized in the expanded online report [11]) ensure that Φ_2 must be the translation of some MixML *usig*.

We feel that Rule 30 is remarkably elegant—certainly it is the most concise formulation of higher-order signature matching that we have ever seen.

6. Related and Future Work

There is a large body of work on ML modules and mixin modules independently, some of which we cited in the introduction. For space reasons, we confine our discussion of related work in this section to modularity mechanisms that attempt a synthesis of ML-style and mixin-style features.

Mixin Modules for ML Duggan and Sourelis [13] were the first to integrate a notion of mixin composition into ML modules with type components. They divide mixin modules into three parts. Of these, only the middle part is "mixable," and it may only contain `datatype` and `fun` bindings. In addition, their focus lies mainly on merging datatype variants and function clauses in order to support extensible datatypes. They consider neither opaque sealing nor hierarchical structures, and expressly disallow separate compilation.

Recursive Modules for ML As mentioned in the introduction, there are several proposals for extending ML with recursive modules [6, 33, 22, 26, 7], but they do not handle separate compilation in general. Both Moscow ML [33] and OCaml [22] support separate compilation for limited classes of recursive modules through the functor mechanism. For example, if recursive modules in these languages do not contain any internal uses of opaque sealing and only contain term components of *pointed* type (*i.e.,* functions or lazy suspensions), they can usually be separately compiled. This covers quite a few common cases, but is not a general solution.

The reason for the restrictions on sealing boils down to the double vision problem. First, none of these languages (with the exception of RMC [7]) properly handles double vision in general. (See Dreyer [7] for details.) Second, even in RMC, if we try to break up a recursive module $\mathtt{rec}\,(\mathrm{X}:\mathit{sig})\;\mathit{mod}$ into separately compiled functors of the form $\lambda(\mathrm{X}:\mathit{sig}).\;\mathit{mod}'$ (where mod' is a substructure of mod), RMC's double vision avoidance mechanism for the `rec` construct does not carry over to the typechecking of the functor construct, so double vision again rears its ugly head.

Ignoring separate compilation, MixML's type system is closely based on RMC's, and our encoding of the `rec` construct for recursive modules yields essentially the same semantics as in RMC.[3] MixML's transparent linking generalizes RMC's `rec` construct, while opaque linking generalizes RMC's sealing operator. This generalization arguably simplifies the semantics of the language: Rules 9 and 10 for transparent and opaque linking are very similar—they match up premise for premise—whereas, in RMC, the rules for recursive and sealed modules differ significantly.

Like MixML, Nakata and Garrigue's *Traviata* language [26] supports definitions of opaquely recursive types, but not transparently recursive ones. Their encoding of opaquely recursive types is simpler than ours in that they do not need to insert `new[·]` as described in Section 3. This is only because their type system does not attempt to avoid double vision in the first place, so there is no need to manually override any double vision avoidance mechanism.

Units Units were originally proposed by Flatt and Felleisen [14] as a recursive module extension to Scheme, which they extended with support for abstract type components. Later work by Owens and Flatt extended units with hierarchical namespaces (called *modules*) and translucent type components [30]. Like MixML, the system presented in the latter paper (hereafter, OF) provides units as a form of mixin module that may contain type components and nested structures, but excludes overriding. Units in OF are first-class, and thus higher-order as well. As such, they subsume the higher-order extension to MixML we described in Section 5, but at the expense of introducing subtyping into the core language.

In OF, as in other mixin-based languages, units may be recursively linked with each other, but they are not hierarchically composable into other units. In contrast, MixML modules are both hierarchically composable and recursively linkable. MixML *units* (named in homage to Flatt's units), which are just suspended modules, are composable both hierarchically and recursively as well. For example, to recursively link units U_1 and U_2 we write $\mathtt{[new}\;\mathrm{U}_1\;\mathtt{with\;new}\;\mathrm{U}_2\mathtt{]}$, as seen at the end of Section 2.

OF requires substantially more bookkeeping annotations from the programmer than MixML. In particular, every unit and linking expression includes explicit specifications of all its imports and exports, and all wiring needed in a linking step must be spelled out explicitly. While this may offer some added flexibility, it becomes extremely burdensome for encoding ML-style modules. Specifically, OF shows how to encode an ML-like module system, but it is one in which modules require signature annotations on essentially every subterm (for example, each functor application involves three distinct signature annotations). Moreover, it is not clear how a general recursive module construct $\mathtt{rec}\,(\mathrm{X}:\mathit{sig})\;\mathit{mod}$ would be expressed in OF. In contrast, our MixML encoding of ML-style modules is simple and direct and includes recursive modules.

Recursive DLLs Duggan [12] presents a language for *recursive DLLs* similar to OF units. His units (called modules in his paper) are enriched with explicit support for sealing and an orthogonal construct for dynamic typing. As in other mixin approaches, his modules are not hierarchically composable. In addition, his system does not support transparent type definitions, only opaque datatype definitions and sharing constraints between abstract types. Like us, Duggan builds compound structures from atomic forms, but to do this he uses a concatenation operator separate from mixin linking.

Signature Operators Ramsey *et al.* [31] propose a variety of extensions to the ML signature language. Some of them are expressible in MixML: signature composition (`andalso`) directly corresponds to linking, and the `adding` and `revealing` constructs for signature extension and refinement can also be encoded using linking. Moreover, they propose a binder (`as`) that plays a role similar to the variable binding in MixML's linking construct. Other extensions presented in their paper, such as renaming and removal of components, cannot be encoded directly in MixML. However, similar operators are present in classical CMS-style mixin modules [3], and we believe these could be readily incorporated into MixML.

Scala Scala is a language developed by Odersky *et al.* [28, 29] combining object-oriented mixin class composition with ML-style type components. It provides several linking operations; in particular, its mixin composition is very similar to our transparent linking, except that it allows overriding and restricts specialization of abstract fields to be left-to-right. The inheritance mechanism can also be viewed as a form of transparent linking, combined with typical OO-style data abstraction via access modifiers on class fields. This form of data abstraction is less expressive than ML-style sealing, which we model using opaque linking. Scala classes are not hierarchically composable in the manner of MixML modules.

An interesting restriction in Scala is that mixin composition is only allowed with a concrete class, not a class represented by an abstract type (*i.e.,* an import). To circumvent this restriction, Scala introduces *selftype annotations*, which can be viewed as a way of allowing recursive linking against as-yet-undefined classes. However, they also delay a certain amount of typechecking, which is only performed when the class is finally instantiated. In contrast, in the higher-order extension of MixML (discussed in Section 5),

[3] We say "essentially" because MixML is more liberal than ML in certain respects. For example, when matching a structure against a signature with a transparent type spec `type t = int`, ML will require the structure to have a type component `t` equal to `int`; MixML will require only that, if the structure does have a `t` component, then it is equal to `int`. While we view this departure from ML as a potentially useful feature, it makes formal comparisons of expressiveness difficult.

unit signatures enable one to specify a unit as an import component of a module, and then, within that module, to instantiate the unit (via `new`) and link it freely with other modules.

The success of Scala was a major impetus for us to figure out how to incorporate mixin composition into the ML module system.

J& Also in the context of object-oriented programming, Nystrom *et al.* [27] argue that *nested intersection* (hierarchical composability) for nested classes is an essential feature for supporting compositional modular extensions. They devise J&, a mixin extension to a Java-like language that supports this feature. Unlike Scala or MixML, the language does not support type components. Consequently, it does not provide ML-style sealing either. Nested class definitions can simulate type components to a certain extent, but are unable to express type equivalences. Thus, it is unclear how J& could encode functor-style modular abstractions involving type sharing specifications.

Future Work We have built a prototype interpreter for a language based on MixML, which includes built-in support for a number of the encodings given in Section 2. It is available for download [11].

We believe that our design already provides a fairly complete basis for a practical module system, and we are currently investigating how we might deploy it in a realistic ML compiler. To further expand its utility, though, we are interested in extending it with support for first-class units (*e.g.,* in the style of Dreyer *et al.*'s modules-as-first-class-values [10], or Rossberg's packages [32]), as well as OCaml-style applicative functors [19], among other features.

References

[1] D. Ancona, S. Fagorzi, E. Moggi, and E. Zucca. Mixin modules and computational effects. In *ICALP '03*.

[2] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[3] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA '90*.

[5] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *ICCL '92*.

[6] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI '99*.

[7] Derek Dreyer. A type system for recursive modules. In *ICFP '07*.

[8] Derek Dreyer. A type system for well-founded recursion. In *POPL '04*.

[9] Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, 17(4&5):433–471, 2007.

[10] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL '03*.

[11] Derek Dreyer and Andreas Rossberg. MixML (project website). http://www.mpi-sws.mpg.de/~rossberg/mixml/.

[12] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems*, 24(6):711–804, 2002.

[13] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *ICFP '96*.

[14] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI '98*.

[15] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94*.

[16] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[17] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems*, 27(5):857–881, 2005.

[18] Mark P. Jones. Using parameterized signatures to express modular structure. In *POPL '96*.

[19] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*.

[20] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94*.

[21] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[22] Xavier Leroy. A proposal for recursive modules in Objective Caml, 2003. http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf.

[23] David MacQueen. Modules for Standard ML. In *LFP '84*.

[24] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[25] D. A. Moon. Object-oriented programming with Flavors. In *OOPSLA '86*.

[26] Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *ICFP '06*.

[27] Nathaniel Nystrom, Xin Qi, and Andrew Myers. J&: Nested intersection for scalable software composition. In *OOPSLA '06*.

[28] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP '03*.

[29] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05*.

[30] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06*.

[31] Norman Ramsey, Kathleen Fisher, and Paul Govereau. An expressive language of signatures. In *ICFP '05*.

[32] Andreas Rossberg. The missing link – dynamic components for ML. In *ICFP '06*.

[33] Claudio V. Russo. Recursive structures for Standard ML. In *ICFP '01*.